

Referência a Linguagem ECL

Equipe de documentação de Boca Raton



Referência a Linguagem ECL

Equipe de documentação de Boca Raton

Sua opinião e comentários sobre este documento são muito bem-vindos e podem ser enviados por e-mail para <docfeedback@hpccsystems.com>, estando sujeitos ao Acordo de Sugestões do HPCC disponível em: hpccsystems.com/contribution. Inclua a frase **Feedback sobre documentação** inclua a frase Feedback sobre documentação na linha de assunto e indique o nome do documento, o número das páginas e número da revisão atual no corpo da mensagem.

A LexisNexis e os logotipos, designs, imagem comercial e marcas registradas relacionados são de propriedade da Reed Elsevier Properties Inc. e de suas afiliadas, são usados sob licença e não estão sujeitos à licença Creative Commons. Outras marcas registradas são de propriedade de suas respectivas empresas e não estão sujeitas à licença Creative Commons.

Todos os nomes e dados de exemplo usados neste manual são fictícios. Qualquer semelhança com pessoas reais, vivas ou mortas, é mera coincidência.

ESTE TRABALHO É FORNECIDO DE ACORDO COM OS TERMOS DA LICENÇA PÚBLICA CREATIVE COMMONS DESCRITOS EM APÊNDICE A (VEJA).

2022 Version 8.4.64-1

Introdução	8
Estrutura da Documentação	8
Convenções de Documentação	9
ECL Básico	10
Visão geral	10
Constantes	11
Definições	14
Tipos Básicos de Definições	16
Filtragem de Recordset	19
Definição de Funções (Passagem de Parâmetros)	20
Visibilidade das definições	25
Qualificação e Definição de Campo	27
Ações e Definições	29
Expressões e Operadores	30
Expressões e Operadores	30
Operadores Lógicos	32
Operadores de Recordset	33
Operadores Set	35
Operadores String	36
Operador IN	37
Operador BETWEEN	38
Tipos de valores	39
BOOLEAN	39
INTEGER	40
REAL	41
DECIMAL	42
STRING	43
QSTRING	44
UNICODE	45
UTF8	46
DATA	47
VARSTRING	48
VARUNICODE	49
SET OF	50
TYPEOF	51
RECORDOF	52
ENUM	53
Conversão de Conversão	54
Estruturas de registros e arquivos	57
Estrutura RECORD	57
DATASET	67
DICTIONARY	83
INDEX	85
Escopo e Nomes de Arquivo Lógicos.	88
Racionalidade Implícitos de Dataset	91
Tipos de Dados Não Nativos	92
Estrutura TYPE	92
Funções Especiais da Estrutura TYPE	93
Suporte a Análise (Parsing)	95
Suporte a Análise (Parsing)	95
Tipo de Valores PARSE Pattern	96
Funções NLP RECORD e TRANSFORM	100
Funções RECORD e TRANSFORM e para Análise de XML	102
Palavras-chave reservadas	104

ALL	104
EXCEPT	105
EXPORT	106
Palavra-chave GROUP	107
IMPORT	108
KEYED e WILD	110
LEFT e RIGHT	112
LIKELY e UNLIKELY	113
ROWS(LEFT) e ROWS(RIGHT)	114
SELF	115
SHARED	116
SKIP	117
TRUE e FALSE	118
Estruturas Especiais	119
Estrutura BEGINC++	120
Estrutura EMBED	125
Estrutura FUNCTION	127
Estrutura FUNCTIONMACRO	130
Estrutura INTERFACE	132
Estrutura MACRO	135
Estrutura MODULE	137
Estrutura TRANSFORM	140
Ações e Funções Built-in	144
ABS	145
ACOS	146
AGGREGATE	147
ALLNODES	150
APPLY	151
ASCII	152
ASIN	153
ASSERT	154
ASSTRING	156
ATAN	157
ATAN2	158
AVE	159
BUILD	160
CASE	166
CATCH	167
CHOOSE	169
CHOOSEEN	170
CHOOSESETS	171
CLUSTERSIZE	172
COMBINE	173
CORRELATION	176
COS	178
COSH	179
COUNT	180
COVARIANCE	182
CRON	184
DEDUP	185
DEFINE	188
DENORMALIZE	189
DISTRIBUTE "Randômico"	192
DISTRIBUTED	195

DISTRIBUTION	196
EBCDIC	198
ENTH	199
ERROR	200
EVALUATE	201
EVENT	203
EVENTNAME	204
EVENTEXTRA	205
EXISTS	206
EXP	207
FAIL	208
FAILCODE	209
FAILMESSAGE	210
FETCH	211
FROMJSON	213
FROMUNICODE	214
FROMXML	215
GETENV	216
GLOBAL	217
GRAPH	218
GROUP	220
HASH	222
HASH32	223
HASH64	224
HASHCRC	225
HASHMD5	226
HAVING	227
HTTPCALL	228
IF	230
IFF	231
IMPORT	232
INTFORMAT	233
ISVALID	234
ITERATE	235
JOIN	237
KEYDIFF	246
KEYPATCH	247
KEYUNICODE	249
LENGTH	250
LIBRARY	251
LIMIT	253
LN	255
LOADXML	256
LOCAL	258
LOG	259
LOOP	260
MAP	262
MAX	263
MERGE	264
MERGEJOIN	266
MIN	268
NOLOCAL	269
NONEMPTY	270
NORMALIZE	271

NOFOLD	274
NOTHOR	275
NOTIFY	276
ORDERED	277
OUTPUT	278
PARALLEL	288
PARSE	289
PIPE	295
POWER	297
PRELOAD	298
PROCESS	299
PROJECT	301
PULL	305
RANDOM	306
RANGE	307
RANK	308
RANKED	309
REALFORMAT	310
REGEXFIND	311
REGEXFINDSET	312
REGEXREPLACE	313
REGROUP	314
REJECTED	316
ROLLUP	317
ROUND	321
ROUNDUP	322
ROW	323
ROWDIFF	327
SAMPLE	328
SEQUENTIAL	329
SET	330
SIN	332
SINH	333
SIZEOF	334
SOAPCALL	335
SORT	339
SORTED	343
SQRT	344
STEPPED	345
STORED	347
SUM	348
TABLE	349
TAN	351
TANH	352
THISNODE	353
TOJSON	354
TOPN	355
TOUNICODE	356
TOXML	357
TRACE	358
TRANSFER	360
TRIM	361
TRUNCATE	362
UNGROUP	363

UNICODEORDER	364
UNORDERED	365
VARIANCE	366
WAIT	368
WHEN	369
WHICH	370
WORKUNIT	371
XMLDECODE	372
XMLENCODE	373
Serviços de Fluxo de Trabalho	374
Visão Geral do Fluxo de Trabalho	375
CHECKPOINT	376
DEPRECATED	377
FAILURE	378
GLOBAL - Serviço	379
INDEPENDENT	380
ONWARNING	381
PERSIST	382
PRIORITY	384
RECOVERY	385
STORED - Serviço de Fluxo de Trabalho	386
SUCCESS	388
WHEN	389
Linguagem Template	390
Visão Geral da Linguagem Template	390
#APPEND	391
#CONSTANT	392
#DECLARE	393
#DEMANGLE	394
#ERROR	395
#EXPAND	396
#EXPORT	397
#EXPORTXML	400
#FOR	402
#GETDATATYPE	403
#IF	404
#INMODULE	405
#LOOP / #BREAK	406
#MANGLE	407
#ONWARNING	408
#OPTION	409
#SET	419
#STORED	420
#TEXT	421
#UNIQUENAME	422
#WARNING	424
#WEBSERVICE	425
#WORKUNIT	426
Serviços Externos	427
Estrutura SERVICE	427
CONST	429
Implementação de Serviços Externos	430
A. Licença Creative Commons	437
Index	441

Introdução

Estrutura da Documentação

Este manual documenta a Enterprise Control Language (ECL). A ECL foi projetada especificamente para trabalhar com grandes conjuntos de dados. Este livro foi projetado para ser uma ferramenta de aprendizado e um trabalho de referência e está dividido nas seguintes seções:

ECL Básico	Aborda os conceitos fundamentais de ECL.
Expressões e Operadores	Define os operadores disponíveis e sua precedência de avaliação de expressão.
Tipos de valores	Introduz os tipos de dados e conversões de tipo.
Estruturas de registros e arquivos	Introduz as estruturas RECORD, DATASET e INDEX
Tipos de dados não nativos	Define a estrutura TYPE e as funções que ela possa vir a usar.
Suporte de análise de linguagem natural	Define os padrões e as funções que a função PARSE possa vir a usar.
Palavras-chave reservadas	Define o uso especial de palavras-chave ECL que não foi definido em nenhum outro lugar.
Estruturas Especiais	Define TRANSFORM, MACRO, e outras estruturas, assim como seu uso.
Ações e Funções Built-in	Define as funções e ações disponíveis como parte da linguagem.
Workflow Services	Define a execução do job/os aspectos de controle do processo ECL.
Templates	Define os comandos do modelo ECL.
External Services	Define a estrutura SERVICE e o seu uso.

Convenções de Documentação

Caso de sintaxe ECL

Embora o ECL não faça distinção entre maiúsculas e minúsculas, as palavras-chave reservadas do ECL e as funções contidas neste documento são sempre exibidas com todos os caracteres EM MAIÚSCULA (ALL CAPS) para dar destaque e facilitar a identificação. A definição e os nomes do conjunto de registros são sempre exibidos no código de exemplo em caracteres maiúsculos e minúsculos misturados. Palavras em execução podem ser usadas para identificar explicitamente a finalidade nos exemplos.

Itens Opcionais

Palavras-chave e parâmetros de uso opcional são incluídos em nos diagramas de sintaxe com as opções e/ou separadas por uma barra vertical (|), como neste exemplo:

EXAMPLEFUNC(*parameter* [,*optionalparameter*] [,OPTIONAL | WORD])

Exemplo:

Todos os códigos de exemplo neste documento são mostrados como na lista abaixo:

```
TotalTrades := COUNT(Trades); // TotalTrades is the Definition name
// COUNT is a built-in function, Trades is the name of a record set
```

ECL Básico

Visão geral

Enterprise Control Language (ECL) foi concebida especificamente para grandes projetos de dados que usam o High Performance Computer Cluster (HPCC) da LexisNexis. A extrema escalabilidade da ECL decorre de uma concepção que permite reutilizar todas as consultas criadas em consultas subsequentes, conforme a necessidade. Para fazer isso, a ECL adota uma abordagem de Dicionário na criação de consultas. Cada definição de ECL define uma expressão. Cada Definição anterior pode ser usada em definições posteriores da ECL – *a linguagem estende a si mesma à medida que é usada*.

Definições versus Ações

Funcionalmente, há dois tipos de código de ECL: Definições (AKA definições de atributo) e Ações executáveis. As Ações não podem ser usadas em expressões porque não retornam valores. A maioria do código de ECL é composta de definições.

As definições definem apenas *o que* deve ser feito, mas não executam nada. Isso significa que o programador de ECL deve pensar em termos de criar código que especifica *o que* fazer em vez de *como* fazer. Esse é um conceito importante na medida em que o programador diz ao supercomputador *o que* precisa acontecer e não *como* o resultado será conseguido. Assim, o supercomputador fica livre para otimizar a execução real da forma necessária para gerar o resultado desejado.

Uma segunda consideração é que a ordem em que as Definições aparecem no código fonte não define a ordem de execução. A ECL é uma linguagem não procedural. Quando uma Action (como OUTPUT) executa, todas as Definitions que precisa usar (considerando até as Definitions de nível mais baixo nas quais as outras são baseadas) são compiladas e otimizadas. Ou seja, ao contrário de outras linguagens de programação, não há uma ordem de execução inerente implícita na ordem em que as definições aparecem no código fonte (embora exista uma ordem necessária para que não corram erros de compilação; referências antecipada referências não são permitidas). Esse conceito de "execução sem ordem" exige uma atitude diferente da usada em linguagens de programação padrão, dependentes de ordem, porque aparenta executar "tudo ao mesmo tempo".

Problemas de Sintaxe

A ECL não faz distinção entre maiúsculas e minúsculas. Os espaços em branco são ignorados, o que permite a formatação necessária para melhor legibilidade.

A ECL admite comentários no código. Os comentários em bloco devem ser delimitados com `/*` e `*/`.

```
/* this is a block comment - the terminator can be on the same line  
or any succeeding line -- everything in between is ignored */
```

Comentários de uma única linha devem começar com `//`.

```
// this is a one-line comment
```

A ECL usa a sintaxe *object.property* padrão utilizada por várias outras linguagens de programação (no entanto, a ECL não é uma linguagem orientada a objeto) para qualificar o escopo das Definições e eliminar ambiguidades das referências de campos dentro de tabelas:

```
ModuleName.Definition //reference an definition from another module/folder
```

```
Dataset.Field //reference a field in a dataset or recordset
```

Constantes

String

Todas as literais de string devem estar entre apóstrofes ('). Todo o código da ECL é codificado em UTF-8, o que significa que todas as strings também são codificadas em UTF-8, tanto as strings Unicode quanto as não Unicode. Portanto, você deve usar um editor UTF-8 (como o programa ECL IDE).

Para incluir o caractere apóstrofo em uma string de constante, acrescente uma barra invertida (\) antes dele. Para incluir um caractere de barra invertida (\) em uma string de constante, use duas barras invertidas juntas (\\).

```
STRING20 MyString2 := 'Fred\'s Place';  
                //evaluated as: "Fred's Place"  
STRING20 MyString3 := 'Fred\\Ginger\'s Place';  
                //evaluated as: "Fred\Ginger's Place"
```

Os outros caracteres de escape disponíveis são:

\t	tab
\n	nova linha
\r	retorno
\nnn	3 dígitos octais (para qualquer outro caractere)
\uhhhh	"u" minúsculo seguido por 4 dígitos hexadecimais (para qualquer outro caractere exclusivamente UNICODE)

```
MyString1 := 'abcd';  
MyString2 := U'abcd\353';    // becomes 'abcdë'
```

As hexadecimais devem começar com um caractere "x". A string de caracteres pode conter apenas valores hexadecimais válidos (0 a 9, A a F) e o número de caracteres deve ser par.

```
DATA2 MyHexString := x'0D0A'; // a 2-byte hexadecimal string
```

As de string de dados devem começar com um caractere "D". Isso é o equivalente direto de converter a constante de string em DATA.

```
MyDataString := D'abcd'; // same as: (DATA)'abcd'
```

As constantes de string Unicode string devem começar com um caractere "U". Os caracteres entre aspas são codificados em utf8 e o tipo de constante é UNICODE.

```
MyUnicodeString1 := U'abcd';    // same as: (UNICODE)'abcd'  
MyUnicodeString2 := U'abcd\353'; // becomes 'abcdë'  
MyUnicodeString3 := U'abcd\u00EB'; // becomes 'abcdë'
```

As constantes de UTF8 string devem começar com os caracteres "U8". Os caracteres entre aspas são codificados em utf8 e o tipo de constante é UTF8.

```
MyUTF8String := U8'abcd\353';
```

As constantes de string VARSTRING devem começar com um caractere "V". O byte nulo de finalização é implícito e o tipo da constante é VARSTRING.

```
MyVarString := V'abcd'; // same as: (VARSTRING)'abcd'
```

As constantes de string QSTRING devem começar com um caractere "Q". O byte nulo de finalização é implícito e o tipo da constante é VARSTRING.

```
MyQString := Q'ABCD'; // same as: (QSTRING)'ABCD'
```

Numéricas

Constantes Numéricas com parte decimal são tratadas como valores REAL (é permitido usar notação científica) e as sem parte decimal são tratadas como INTEGER (consulte **Tipos de valores**). As constantes inteiras podem ser valores decimais, não-definidos, hexadecimais ou binários. Valores hexadecimais são especificados começando com "0x" ou terminando com um caractere "x". Valores binários são especificados começando com "0b" ou terminando com um caractere "b". Valores Decimais são especificados com o caractere "d" à direita. Valores não-definidos são especificados com o caractere "u" à direita.

```
MyInt1 := 10; // value of MyInt1 is the INTEGER value 10
MyInt2 := 0x0A; // value of MyInt2 is the INTEGER value 10
MyInt3 := 0Ax; // value of MyInt3 is the INTEGER value 10
MyInt4 := 0b1010; // value of MyInt4 is the INTEGER value 10
MyInt5 := 1010b; // value of MyInt5 is the INTEGER value 10
MyReal1 := 10.0; // value of MyReal1 is the REAL value 10.0
MyReal2 := 1.0e1; // value of MyReal2 is the REAL value 10.0
MyDec1 := 10d // value of MyDec1 is the DECIMAL value 10
MyDec2 := 3.14159265358979323846d // value of MyDec2 is the DECIMAL
// value 3.14159265358979323846
// a REAL type would lose precision
```

Compilar Constantes de Tempo

As seguintes constantes de sistema constantes estão disponíveis no momento da compilação. Essas constantes podem ser úteis na criação de código condicional.

<code>__ECL_VERSION__</code>	Uma STRING que contém o valor da versão da plataforma. Por exemplo, "6.4.0"
<code>__ECL_VERSION_MAJOR__</code>	Um INTEGER que contém o valor da parte principal da versão da plataforma. Por exemplo, "6"
<code>__ECL_VERSION_MINOR__</code>	Um INTEGER que contém o valor da parte secundária da versão da plataforma. Por exemplo, "4"
<code>__ECL_LEGACY_MODE__</code>	Um valor BOOLEAN indicando se a compilação está sendo executada com semântica de IMPORT legada.
<code>__OS__</code>	Uma STRING indicando o sistema operacional de destino da compilação. Os valores possíveis são: 'Windows', 'MacOS', ou 'linux'.
<code>__STAND_ALONE__</code>	Um valor BOOLEAN indicando se a compilação gerará um executável autônomo.
<code>__TARGET_PLATFORM__</code>	Uma STRING contendo o valor da plataforma de destino (o tipo de cluster em que a consulta foi submetida). Os valores possíveis são: 'roxie', 'hthor', 'thor', or 'thorlcr'.
<code>__PLATFORM__</code>	Uma STRING contendo o valor da plataforma em que a consulta será executada. Os valores possíveis são: 'roxie', 'hthor', 'thor', or 'thorlcr'.
<code>__CONTAINERIZED__</code>	Um valor BOOLEANO que indica se a plataforma é uma versão em contêiner.

Exemplo:

```
IMPORT STD;
STRING14 fGetDateTimeString() :=
  #IF(__ECL_VERSION_MAJOR__ > 5) or ((__ECL_VERSION_MAJOR__ = 5)
    AND (__ECL_VERSION_MINOR__ >= 2))
    STD.Date.SecondsToString(STD.Date.CurrentSeconds(true), '%Y%m%d%H%M%S');
  #ELSE
    FUNCTION
      string14 fGetDimeTime():= // 14 characters returned
      BEGINC++
      #option action
      struct tm localtime;      // localtime in "tm" structure
      time_t timeinsecs;        // variable to store time in secs
      time(&timeinsecs);
      localtime_r(&timeinsecs,&localt);
      char temp[15];
      strftime(temp, 15, "%Y%m%d%H%M%S", &localt);
      // Formats the localtime to YYYYMMDDhhmmss
      strncpy(__result, temp, 14);
      ENDC++;
      RETURN fGetDimeTime();
    END;
  #END;
```

Definições

Cada Definição ECL é o componente básico da ECL. Uma definição especifica *o que* fazer, mas não *como* fazer. As definições podem ser consideradas como uma forma altamente desenvolvida de substituição por macro, em que cada definição sucessiva aproveita cada vez mais o trabalho realizado anteriormente. O resultado é uma construção de query (consulta) extremamente eficiente.

Todas as definições têm o seguinte formato:

`[Scope] [ValueType] Name [(parms)] := Expression [:WorkflowService] ;`

The Definition Operator (`:=` read as “is defined as”) defines an expression. No lado esquerdo do operador, há um *Escopo* opcional (consulte **Visibilidade de atributos**), um *Tipo de Valores* (consulte **Tipos de valores**) e os parâmetros (*parms*) que ele pode receber (consulte **Funções (Passagem de parâmetros)**). No lado direito, a expressão que produz o resultado e, opcionalmente, dois pontos (`:`) e uma lista de *WorkflowServices* (consulte **Serviços de fluxo de trabalho**) delimitada por vírgulas. Uma definição deve ser concluída explicitamente com ponto e vírgula (`;`). O nome da Definição pode ser usado em definições subsequentes:

```
MyFirstDefinition := 5; //defined as 5
MySecondDefinition := MyFirstDefinition + 5; //this is 10
```

Regras de nomenclatura de definição

Os nomes de Definições s válidos começam com uma letra e podem conter apenas letras, números e o caractere sublinhado (`_`).

```
My_First_Definition1 := 5; // valid name
My First Definition := 5; // INVALID name, spaces not allowed
```

Você poderá atribuir a uma Definição o nome de um módulo criado anteriormente no Repositório do ECL se o atributo estiver definido com um *ValueType*.

Palavras Reservadas

As palavras-chave do ECL, as funções integradas e suas opções são palavras reservadas , mas de maneira geral apenas no contexto em que seu uso é válido. Mesmo nesse contexto, você pode usar palavras reservadas como nomes de campos ou definições, desde que elimine explicitamente qualquer ambiguidade, como neste exemplo:

```
ds2 := DEDUP(ds, ds.all, ALL); //ds.all is the 'all' field in the
                               //ds dataset - not DEDUP's ALL option
```

No entanto, é uma boa ideia evitar usar as palavras-chave da ECL como nomes de definições ou campos.

Os nomes de definições ou campos não pode começar com **UNICODE_**, **UTF8_**, ou **VARUNICODE_**. Os rótulos que começam com esses prefixos são tratados como nomes de tipo e devem ser considerados como reservados.

Nomenclatura de Definição

Use nomes descritivos para todas as definições **EXPORTed** e **SHARED**. Dessa forma, o código ficará mais legível. A convenção de nomenclatura adotada em toda a documentação e em todos os cursos de treinamento da ECL é a seguinte:

Definition Type	Are Named
Boolean	Is...
Set Definition	Set...
Record Set	...DatasetName

Por exemplo:

```
IsTrue := TRUE;           // a BOOLEAN Definition
SetNumbers := [1,2,3,4,5]; // a Set Definition
R_People := People(firstname[1] = 'R'); // a Record Set Definition
```

Tipos Básicos de Definições

Os tipos básicos de definições usados com mais frequência na programação em ECL são: **Boolean**, **Value**, **Set**, **Record**, **Set**, e **TypeDef**.

Definição BOOLEANS

Uma definição booleana é definida como qualquer Definição cuja definição seja uma expressão lógica com resultado TRUE/FALSE (VERDADEIRO/FALSO). Por exemplo, todas as definições a seguir são booleanas:

```
IsBoolTrue := TRUE;
IsFloridian := Person.per_st = 'FL';
IsOldPerson := Person.Age >= 65;
```

Definição Values

Uma Value é definida como qualquer definição cuja expressão seja aritmética ou de string com um resultado de valor único. Por exemplo, todos os seguintes são definições de valor:

```
ValueTrue := 1;
FloridianCount := COUNT(Person(Person.per_st = 'FL'));
OldAgeSum := SUM(Person(Person.Age >= 65), Person.Age);
```

Definições Set

Uma definição SET é definida como qualquer definição cuja expressão seja um conjunto de valores expressados em colchetes. Conjunto de constantes são criados como um conjunto de valores constantes declarados de forma explícita que precisam ser expressados entre colchetes, seja esse conjunto definido como uma definição separada ou simplesmente incluída em linha em outra expressão. Todas as constantes precisam ser do mesmo tipo.

```
SetInts := [1,2,3,4,5]; // an INTEGER set with 5 elements
SetReals := [1.5,2.0,3.3,4.2,5.0];
// a REAL set with 5 elements
SetStatusCodes := ['A','B','C','D','E'];
// a STRING set with 5 elements
```

Os elementos em qualquer conjunto declarado de forma explícita também podem ser compostos de expressões arbitrárias. Todas as expressões precisam resultar no mesmo tipo e precisam ser expressões constantes.

```
SetExp := [1,2+3,45,SomeIntegerDefinition,7*3];
// an INTEGER set with 5 elements
```

Sets podem conter definições e expressões, além de constantes, contanto que todos os elementos sejam do mesmo tipo de resultado. Por exemplo:

```
StateCapitol(STRING2 state) :=
CASE(state, 'FL' => 'Tallahassee', 'Unknown');
SetFloridaCities := ['Orlando', StateCapitol('FL'), 'Boca '+ 'Raton',
person[1].per_full_city];
```

Set Definitions também podem ser definidas usando a função SET (consulte) SETS definidos desta maneira podem ser usados como qualquer outro SET

```
SetSomeField := SET(SomeFile, SomeField);
// a set of SomeField values
```

SETs também podem conter datasets para uso com essas funções (como: MERGE, JOIN, MERGEJOIN ou GRAPH), exigindo conjuntos de datasets como parâmetros de entrada.


```
SetDS := [ds1, ds2, ds3]; // a set of datasets
```

Você pode construir um DATASET a partir from a SET.

```
SET OF STRING s := ['Jim','Bob','Richard','Tom'];  
DATASET(s,{STRING txt});
```

Ordenação e Indexação de Set

SETs são ordenados de forma implícita e você pode indexá-los para acessar elementos individuais. Colchetes são usados para especificar o número do elemento para acesso. O primeiro elemento é o número um (1).

```
MySet := [5,4,3,2,1];  
ReverseNum := MySet[2]; //indexing to MySet's element number 2,  
                        //so ReverseNum contains the value 4
```

Strings (SET de caracteres) podem também ser indexados para acessar elementos contíguos individuais ou múltiplos dentro do SET de caracteres (uma string é tratada como se fosse um conjunto de strings de 1 caractere). Um número de elemento dentro de colchetes especifica um caractere individual a ser extraído.

```
MyString := 'ABCDE';  
MySubString := MyString[2]; // MySubString is 'B'
```

Substrings podem ser extraídas usando dois pontos finais para separar os números de elemento inicial e final nos colchetes, a fim de especificar a (string slice) a ser extraída. O número do elemento inicial ou final pode ser omitido para indicar uma substring do início ao elemento especificado, ou do elemento especificado até o final.

```
MyString := 'ABCDE';  
MySubString1 := MyString[2..4]; // MySubString1 is 'BCD'  
MySubString2 := MyString[ ..4]; // MySubString2 is 'ABCD'  
MySubString3 := MyString[2.. ]; // MySubString3 is 'BCDE'
```

Definição Recordset

O termo "Dataset" no ECL significa explicitamente um arquivo de dados "físico" no supercomputador (em disco ou na memória), enquanto o termo "Record Set" indica qualquer conjunto de registros derivados de um dataset (ou outro Record Set) normalmente baseado em alguma condição de filtro para limitar o conjunto de resultados para um subconjunto de registros. Conjuntos de registros também são criados como o resultado de retorno de uma das funções integradas que retornam conjuntos de resultados.

Uma definição de Record Set é definida como qualquer definição cuja expressão seja um Record Set ou dataset filtrado, ou qualquer função que retorne um Record Set. Por exemplo, todos os itens a seguir são Definições de conjunto de registro:

```
FloridaPersons := Person(Person.per_st = 'FL');  
OldFloridaPersons := FloridaPersons(Person.Age >= 65);
```

Ordenação e Indexação do Recordset

Todos os Datasets e Conjuntos de registro são organizados de forma implícita e podem ser indexados para acessar registros individuais dentro do conjunto. Colchetes são usados para especificar o número de elemento para acesso, e o primeiro elemento em qualquer conjunto é o número um (1).

Datasets (incluindo datasets filhos) e Conjuntos de registro podem usar o mesmo método descrito acima para strings para acessar registros contíguos individuais ou múltiplos.

```
MyRec1 := Person[1]; // first rec in dataset  
MyRec2 := Person[1..10]; // first ten recs in dataset
```

```
MyRec4 := Person[2..]; // all recs except the first
```

Observação: ds[1] e ds[1..1] não são idênticos - ds[1..1] é um Record Set (pode ser usado no contexto de Record Set), enquanto ds[1] é uma linha única (pode ser usada para referenciar campos únicos).

E você também pode acessar campos individuais em um registro específico com um índice único:

```
MyField := Person[1].per_last_name; // last name in first rec
```

A indexação de um Record Set com um valor que esteja fora dos limites é definida para retornar uma linha na qual todos os campos contenham valores zerados/em branco. Muitas vezes, é mais eficiente indexar um valor fora do limite em vez de usar uma programação que processa o caso especial de um valor de índice fora do limite.

Por exemplo, a expressão:

```
IF(COUNT(ds) > 0, ds[1].x, 0);
```

is simpler as:

```
ds[1].x //note that this returns 0 if ds contains no records.
```

Definições TypeDef

Uma definição TypeDef é definida como qualquer definição cuja expressão é um tipo de valor, seja ela integrada ou definida pelo usuário. Por exemplo, todas as definições a seguir são TypeDef (exceto GetXLen):

```
GetXLen(DATA x,UNSIGNED len) := TRANSFER(((DATA4)(x[1..len])),UNSIGNED4);

EXPORT xstring(UNSIGNED len) := TYPE
  EXPORT INTEGER PHYSICALENGTH(DATA x) := GetXLen(x,len) + len;
  EXPORT STRING LOAD(DATA x) := (STRING)x[(len+1)..GetXLen(x,len) + len];
  EXPORT DATA STORE(STRING x):= TRANSFER(LENGTH(x),DATA4)[1..len] + (DATA)x;
END;

pstr := xstring(1); // typedef for user defined type
pppstr := xstring(3);
nameStr := STRING20; // typedef of a system type

namesRecord := RECORD
  pstr surname;
  nameStr forename;
  pppStr addr;
END;
//A RECORD structure is also a typedef definition (user-defined)
```

Filtragem de Recordset

Filtros são expressões condicionais contidas entre parênteses após o nome do Dataset ou Conjunto de registros. Várias condições de filtro podem ser especificadas separando cada expressão de filtro com uma vírgula (. Todas as condições de filtro separadas por vírgulas devem ser TRUE para que um registro seja incluído, o que faz da vírgula um operador AND implícito (veja **Operadores lógicos**), apenas neste contexto.

```
MyRecordSet := Person(per_last_name >= 'T', per_last_name < 'U');  
// MyRecordSet contains people whose last name begins with "T"  
// the comma is an implicit AND while also functioning as  
// an expression separator (implicit parentheses)  
  
MyRecordSet := Person(per_last_name >= 'T' AND per_last_name < 'U');  
// exactly the same logical expression as above  
  
RateGE7trds := Trades(trd_rate >= '7');  
  
ValidTrades := Trades(NOT rmsTrade.Mortgage AND  
                      NOT rmsTrade.HasNarrative(rmsTrade.snClosed));
```

BOOLEAN As definições booleanas devem ser usadas como filtros de conjunto de registros para maximizar a flexibilidade, a legibilidade e a reutilização, em vez de codificá-las diretamente em uma definição de Conjunto de registros. Por exemplo, use:

```
IsRevolv := trades.trd_type = 'R'  
          OR (~ValidType(trades.trd_type)  
            AND trades.trd_acct[1] IN ['4', '5', '6']);  
  
isBank := trades.trd_ind_code IN SetBankIndCodes;  
  
IsBankCard := IsBank AND IsRevolv;  
  
WithinDate(INTEGER1 months) := ValidDate(trades.trd_drpt) AND  
                                trades.trd_drpt_mos <= months;  
  
BankCardTrades := trades(isBankCard AND WithinDate(6));
```

em vez de:

```
BankCardTrades := trades(trades.trd_ind_code IN SetBankIndCodes,  
                          (trades.trd_type = 'R' OR  
                           (~ValidType(trades.trd_type) AND  
                            trades.trd_acct[1] IN ['4', '5', '6'])),  
                          ValidDate(trades.trd_drpt),  
                          trades.trd_drpt_mos <= 6);
```

As vírgulas usadas para separar condições de filtro em uma definição de filtro de conjunto de registros atuam como uma operação **AND** implícita e um conjunto de parênteses em volta dos filtros individuais sendo separados. O resultado é uma ligação mais estreita que o uso do AND em vez de uma vírgula sem parênteses. Por exemplo, a expressão de filtro nesta definição:

```
BankMortTrades := trades(isBankCard OR isMortgage, isOpen);
```

é avaliada como foi escrita:

```
(isBankCard OR isMortgage) AND isOpen
```

e não como:

```
isBankCard OR isMortgage AND isOpen
```

Definição de Funções (Passagem de Parâmetros)

Todos os tipos básicos de Definição também podem se tornar funções. Para isso, basta ajustá-las para aceitar passagem de parâmetros (argumentos). O recurso de receber parâmetros não altera a natureza essencial do tipo da Definição, mas simplesmente aumenta a sua flexibilidade.

As definições de parâmetros aparecem sempre entre parênteses, anexadas ao nome da Definição. Você pode definir a função para receber quantos parâmetros forem necessários para criar a funcionalidade desejada. Basta separar cada definição de parâmetro sucessiva com uma vírgula.

O formato das definições de parâmetros é o seguinte:

DefinitionName([*ValueType*] *AliasName* [=*DefaultValue*]) := expression;

<i>ValueType</i>	Opcional. Especifica o tipo dos dados passados. Se omitido, o padrão é INTEGER: (veja Tipos de valores). Isso também pode incluir a palavra-chave CONST (ver CONST) para indicar que o valor passado será tratado sempre como constante.
<i>AliasName</i>	Atribui um nome ao parâmetro para uso na expressão.
<i>DefaultValue</i>	Opcional. Fornece o valor a ser usado na expressão se o parâmetro for omitido. O <i>DefaultValue</i> poderá ser a palavra-chave ALL se o <i>ValueType</i> for SET (consulte a palavra-chave SET) para indicar todos os valores possíveis para esse tipo de conjunto, ou colchetes vazios ([]) para indicar que não há nenhum valor possível para esse tipo de conjunto.
<i>expression</i>	A operação da função que usa os parâmetros.

Parâmetros simples de tipo value

Se *ValueType* opcional for um dos tipos simples (BOOLEAN, INTEGER, REAL, DECIMAL, STRING, QSTRING, UNICODE, DATA, VARSTRING ou VARUNICODE), o *ValueType* poderá incluir a palavra-chave CONST (consulte **CONST**) para indicar que o valor passado será tratado sempre como constante (normalmente, usado apenas em protótipos da ECL para funções externas).

```
ValueDefinition := 15;
FirstFunction(INTEGER x=5) := x + 5;
    //takes an integer parameter named "x" and "x" is used in the
    //arithmetic expression to indicate the usage of the parameter

SecondDefinition := FirstFunction(ValueDefinition);
    // The value of SecondDefinition is 20

ThirdDefinition := FirstFunction();
    // The value of ThirdDefinition is 10, omitting the parameter
```

Parâmetros SET

O *DefaultValue* para parâmetros SET pode ser um conjunto padrão de valores, a palavra-chave ALL para indicar todos os valores possíveis para esse tipo de conjunto ou colchetes vazios ([]) para indicar que não há nenhum valor possível para esse tipo de conjunto (um conjunto vazio).

```
SET OF INTEGER1 SetValues := [5,10,15,20];
```

```
IsInSetFunction(SET OF INTEGER1 x=SetValues,y) := y IN x;

OUTPUT(IsInSetFunction([1,2,3,4],5)); //false
OUTPUT(IsInSetFunction(,5)); // true
```

Passagem de Parâmetros DATASET

A passagem de um DATASET ou de um conjunto de registros derivado como parâmetro pode ser feita usando a seguinte sintaxe:

DefinitionName(**DATASET**(*recstruct*) *AliasName*) := *expression*;

recstruct é obrigatório e atribui um nome à estrutura de RECORD que define o layout dos campos no parâmetro passado DATASET. Como alternativa, *recstruct* pode usar a função RECORDOF. *AliasName* é obrigatório, atribui um nome ao dataset a ser usado na função e é utilizado em *expression* da Definição para indicar em que local da operação o parâmetro passado será usado. Para ver mais exemplos, consulte a discussão sobre **DATASET como Value Type** na documentação de DATASET.

```
MyRec := {STRING1 Letter};

SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],MyRec);

FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A','C','E']);
//passed dataset referenced as "ds" in expression

OUTPUT(FilteredDS(SomeFile));
```

Passagem de Parâmetros DICTIONARY

A passagem de um DICTIONARY como parâmetro pode ser feita usando a seguinte sintaxe:

DefinitionName(**DICTIONARY**(*structure*) *AliasName*) := *expression*;

O parâmetro obrigatório *structure* é a estrutura de RECORD que define o layout dos campos no parâmetro DICTIONARY passado (normalmente, definido em linha). *AliasName* atribui um nome ao DICTIONARY para uso na função e é utilizado em *expression* da Definição para indicar em que local da operação o parâmetro passado será usado. Consulte a discussão sobre **DICTIONARY como Value Type** na documentação de DICTIONARY.

```
rec := RECORD
  STRING10 color;
  UNSIGNED1 code;
  STRING10 name;
END;

Ds := DATASET([{'Black' ,0 , 'Fred'},
               {'Brown' ,1 , 'Seth'},
               {'Red' ,2 , 'Sue'},
               {'White' ,3 , 'Jo'}], rec);

DsDCT := DICTIONARY(DS,{color => DS});

DCTrec := RECORD
  STRING10 color =>
  UNSIGNED1 code,
  STRING10 name,
END;

InlineDCT := DICTIONARY([{'Black' => 0 , 'Fred'},
                         {'Brown' => 1 , 'Sam'},
                         {'Red' => 2 , 'Sue'},
                         {'White' => 3 , 'Jo'} ],
DCTrec);
```

```
MyDCTfunc(DICTIONARY(DCTrec) DCT,STRING10 key) := DCT[key].name;

MyDCTfunc(InlineDCT,'White'); //Jo
MyDCTfunc(DsDCT,'Brown'); //Seth
```

Passagem de Parâmetros Typeless

É possível passar parâmetros de qualquer tipo usando a palavra-chave ANY como o tipo do valor passado:

DefinitionName (ANY AliasName) := expression;

```
a := 10;
b := 20;
c := '1';
d := '2';
e := '3';
f := '4';
s1 := [c,d];
s2 := [e,f];

ds1 := DATASET(s1,{STRING1 ltr});
ds2 := DATASET(s2,{STRING1 ltr});

MyFunc(ANY l, ANY r) := l + r;

MyFunc(a,b); //returns 30
MyFunc(a,c); //returns '101'
MyFunc(c,d); //returns '12'
MyFunc(s1,s2); //returns a set: ['1','2','3','4']
MyFunc(ds1,ds2); //returns 4 records: '1', '2', '3', and '4'
```

Passagem de Parâmetros Function

A passagem de uma Função como tipo de parâmetro pode ser feita usando uma das seguintes opções de sintaxe como *ValueType* do parâmetro:

FunctionName(parameters)

PrototypeName

<i>FunctionName</i>	O nome de uma função cujo tipo pode ser passado como parâmetro.
<i>parameters</i>	As definições de parâmetros para o parâmetro <i>FunctionName</i> .
<i>PrototypeName</i>	O nome de uma função previamente definida para uso como tipo de função que pode ser passado como parâmetro.

O código a seguir oferece exemplos dos dois métodos:

```
//a Function prototype:
INTEGER actionPrototype(INTEGER v1, INTEGER v2) := 0;

INTEGER aveValues(INTEGER v1, INTEGER v2) := (v1 + v2) DIV 2;
INTEGER addValues(INTEGER v1, INTEGER v2) := v1 + v2;
INTEGER multiValues(INTEGER v1, INTEGER v2) := v1 * v2;

//a Function prototype using a function prototype:
INTEGER applyPrototype(INTEGER v1, actionPrototype actionFunc) := 0;

//using the Function prototype and a default value:
```

```
INTEGER applyValue2(INTEGER v1,
                    actionPrototype actionFunc = aveValues) :=
    actionFunc(v1, v1+1)*2;

//Defining the Function parameter inline, witha default value:
INTEGER applyValue4(INTEGER v1,
                    INTEGER actionFunc(INTEGER v1,INTEGER v2) = aveValues)
    := actionFunc(v1, v1+1)*4;
INTEGER doApplyValue(INTEGER v1,
                    INTEGER actionFunc(INTEGER v1, INTEGER v2))
    := applyValue2(v1+1, actionFunc);

//producing simple results:
OUTPUT(applyValue2(1));           // 2
OUTPUT(applyValue2(2));           // 4
OUTPUT(applyValue2(1, addValues)); // 6
OUTPUT(applyValue2(2, addValues)); // 10
OUTPUT(applyValue2(1, multiValues)); // 4
OUTPUT(applyValue2(2, multiValues)); // 12
OUTPUT(doApplyValue(1, multiValues)); // 12
OUTPUT(doApplyValue(2, multiValues)); // 24

//A definition taking function parameters which themselves
//have parameters that are functions...

STRING doMany(INTEGER v1,
              INTEGER firstAction(INTEGER v1,
                                   INTEGER actionFunc(INTEGER v1,INTEGER v2)),
              INTEGER secondAction(INTEGER v1,
                                   INTEGER actionFunc(INTEGER v1,INTEGER v2)),
              INTEGER actionFunc(INTEGER v1,INTEGER v2))
    := (STRING)firstAction(v1, actionFunc) + ':' + (STRING)secondaction(v1, actionFunc);

OUTPUT(doMany(1, applyValue2, applyValue4, addValues));
// produces "6:12"

OUTPUT(doMany(2, applyValue4, applyValue2,multiValues));
// produces "24:12"
```

Passagem de Parâmetros NAMED

Normalmente, a passagem de valores para uma função definida para receber vários parâmetros, muitos dos quais têm valores padrão (e, portanto, podem ser omitidos), é feita "contando vírgulas" para garantir que os valores escolhidos sejam passados ao parâmetro correto, considerando a posição do parâmetro na lista. Esse método não é prático quando há muitos parâmetros opcionais.

O método mais fácil é usar a seguinte sintaxe de parâmetro NAMED, que elimina a necessidade de incluir vírgulas incômodas como espaços reservados para que os valores sejam passados aos parâmetros corretos:

Attr := FunctionName([**NAMED**] *AliasName* := *value*);

<i>NAMED</i>	Opcional. Obrigatório apenas quando <i>AliasName</i> conflita com uma palavra reservada.
<i>AliasName</i>	Os nomes do parâmetro na definição da função da Definição.
<i>value</i>	O valor a ser passado para o parâmetro.

Essa sintaxe é usada na chamada das funções e permite passar valores para parâmetros específicos pelo seu *AliasName*, independentemente de sua posição na lista. Todos os parâmetros passados sem nome atribuído devem preceder todos os parâmetros NAMED.

```
outputRow(BOOLEAN showA = FALSE, BOOLEAN showB = FALSE,
          BOOLEAN showC = FALSE, STRING aValue = 'abc',
          INTEGER bValue = 10, BOOLEAN cValue = TRUE) :=
  OUTPUT(IF(showA, ' a='+aValue, '')+
         IF(showB, ' b='+ (STRING)bValue, '')+
         IF(showC, ' c='+ (STRING)cValue, ''));

outputRow(); //produce blanks
outputRow(TRUE); //produce "a=abc"
outputRow(, TRUE); //produce "c=TRUE"
outputRow(NAMED showB := TRUE); //produce "b=10"

outputRow(TRUE, NAMED aValue := 'Changed value');
//produce "a=Changed value"

outputRow(, , 'Changed value2', NAMED showA := TRUE);
//produce "a=Changed value2"

outputRow(showB := TRUE); //produce "b=10"

outputRow(TRUE, aValue := 'Changed value');
outputRow(, , 'Changed value2', showA := TRUE);
```


Visibilidade das definições

Código ECL e suas definições são armazenados em arquivos .ECL em seu repositório, que são organizados em módulos (diretórios ou pastas em disco). Cada arquivo .ECL pode conter apenas uma definição EXPORT ou SHARED única (consulte abaixo) juntamente com quaisquer definições locais de apoio necessárias para definir completamente o resultado da definição. O nome do arquivo e o nome de sua definição EXPORT ou SHARED precisam coincidir de forma exata.

Em um módulo (diretório ou pasta em disco), você pode ter quantas definições EXPORT e/ou SHARED precisar. Uma declaração IMPORT (veja a **IMPORTAÇÃO** uma declaração IMPORT (consulte a palavra-chave IMPORT) identifica quaisquer outros módulos cujas definições visíveis estarão disponíveis para uso na definição atual.

Os seguintes escopos de visibilidade de definição fundamentais estão disponíveis no ECL: **"Global"**, **Module** e **Local**.

"Global"

As definições expressadas como **EXPORT** (consulte a palavra-chave **EXPORT**) estão disponíveis no módulo em que são definidas e em qualquer outro módulo que realize o **IMPORTs** desse módulo (consulte a palavra-chave IMPORT).

```
//inside the Definition1.ecl file (in AnotherModule folder) you have:
EXPORT Definition1 := 5;
//EXPORT makes Definition1 available to other modules and
//also available throughout its own module
```

Module

O escopo das definições expressadas como **SHARED** (consulte a palavra-chave **SHARED**) é limitado para esse módulo, estando elas disponíveis em todo o módulo (diferentemente das definições locais). Isso permite que você mantenha privadas quaisquer definições que são necessárias apenas para implementar funcionalidades internas. Definições SHARED são usadas para apoiar definições EXPORT.

```
//inside the Definition2.ecl file you have:
IMPORT AnotherModule;
//makes definitions from AnotherModule available to this code, as needed

SHARED Definition2 := AnotherModule.Definition1 + 5;
//Definition2 available throughout its own module, only

//*****
//then inside the Definition3.ecl file (in the same folder as Definition2) you have:
IMPORT $;
//makes definitions from the current module available to this code, as needed

EXPORT Definition3 := $.Definition2 + 5;
//make Definition3 available to other modules and
//also available throughout its own module
```

Local

Uma definição sem as palavras-chave EXPORT ou SHARED só está disponível para definições subsequentes até o final da próxima definição EXPORT ou SHARED. Isso torna essas definições privadas úteis apenas no escopo daquela definição EXPORT ou SHARED, o que permite que você mantenha privadas quaisquer definições que são necessárias apenas para implementar funcionalidades internas. As definições locais (Local) são usadas para apoiar a definição EXPORT ou SHARED nos arquivos em que estejam presentes. As definições locais (Local) são referenciadas apenas por seu nome de definição, sem a necessidade de qualificação.

```
//then inside the Definition4.ecl file (in the same folder as Definition2) you have:
```

```
IMPORT $;
    //makes definitions from the current module available to this code, as needed

LocalDef := 5;
    //local -- available through the end of Definition4's definition, only

EXPORT Definition4 := LocalDef + 5;
//EXPORT terminates scope for LocalDef

LocalDef2 := Definition4 + LocalDef;
    //INVALID SYNTAX -- LocalDef is out of scope here
    //and any local definitions following the EXPORT
    //or SHARED definition in the file are meaningless
    //since they can never be used by anything
```

A palavra-chave **LOCAL** é válida para uso em qualquer estrutura aninhada, mas é mais útil em uma estrutura **FUNCTIONMACRO** para identificar de forma clara que o escopo de uma definição é limitado ao código gerado na **FUNCTIONMACRO**.

```
AddOne(num) := FUNCTIONMACRO
    LOCAL numPlus := num + 1;
    RETURN numPlus;
ENDMACRO;

numPlus := 'this is a syntax error without LOCAL in the FUNCTIONMACRO';
numPlus;
AddOne(5);
```

Ver também: **IMPORT**, **EXPORT**, **SHARED**, **MODULE**, **FUNCTIONMACRO**

Qualificação e Definição de Campo

Definições

Imported definidas dentro de outro módulo e (palavras-chave **EXPORT** e **IMPORT**) estão disponíveis para uso na definição que contém o **IMPORT**. As Definições importadas devem ser totalmente qualificadas com seu nome de Módulo e Definição usando a sintaxe de ponto (módulo.definição).

```
IMPORT abc; //make all exported definitions in the abc module available
EXPORT Definition1 := 5; //make Definition1 available to other modules
Definition2 := abc.Definition2 + Definition1;
// object qualification needed for Definitions from abc module
```

Campos no Datasets

Cada Dataset conta como um escopo qualificado e os campos delas são totalmente qualificados pelo seu nome de Dataset (ou conjunto de registros) e Campo usando a sintaxe de ponto (dataset.campo). Da mesma forma, o conjunto de resultados da função integrada **TABLE** (consulte a palavra-chave **TABLE**) também atua como um escopo qualificado. O nome do conjunto de registros ao qual um campo pertence é o nome do objeto:

```
Young := YearOf(Person.per_dbrth) < 1950;
MySet := Person(Young);
```

Ao atribuir um nome a um Dataset como parte da definição, os campos dessa Definição (ou conjunto de registros) ingressam no escopo. Se Definições parametrizadas (funções) forem aninhadas, apenas o escopo mais interno estará disponível. Ou seja, todos os campos de um Dataset (ou conjunto de registros derivado) estão no escopo na expressão de filtro. O mesmo ocorre para parâmetros de expressões de qualquer função integrada que atribui um nome a um Dataset ou conjunto de registros derivado como um parâmetro.

```
MySet1 := Person(YearOf(dbrth) < 1950);
// MySet1 is the set of Person records who were born before 1950
```

```
MySet2 := Person(EXISTS(OpenTrades(AgeOf(trd_dla) < AgeOf(Person.per_dbrth))));
```

```
// OpenTrades is a pre-defined record set.
//All Trades fields are in scope in the OpenTrades record set filter
//expression, but Person is required here to bring Person.per_dbrth
// into scope
//This example compares each trades' Date of Last Activity to the
// related person's Date Of Birth
```

Qualquer campo em um Record Set pode ser qualificado com o nome do Dataset em que é baseado ou com o nome de qualquer outro Record Set baseado no mesmo dataset. Por exemplo:

```
memtrade.trd_drpt
nondup_trades.trd_drpt
trades.trd_drpt
```

todas as referências são para o mesmo campo no dataset memtrade.

Normalmente, para fins de consistência, você deve usar o nome do dataset base para qualificação. Você também pode usar o nome do Conjunto de registros atual em qualquer contexto em que o nome do dataset base pode gerar confusão.

Operador de Resolução de Escopo

Os identificadores são examinados na seguinte ordem:

1. O dataset ativo no momento, se houver
2. A definição sendo definida e todos os parâmetros em que está baseada
3. Todas as definições ou parâmetros de qualquer estrutura MODULE ou FUNCTION que contém a definição atual

Isso pode significar que o acesso à definição ou ao parâmetro desejado pode não ser realizado porque está oculto como um nome de parâmetro ou definição privada que conflita com o nome de um campo do dataset.

Seria melhor alterar o nome do parâmetro ou da definição privada para impedir o conflito de nomes, mas às vezes isso não é possível.

Você pode direcionar o acesso a uma correspondência diferente qualificando o nome do campo com o operador de resolução de escopo (o caractere circunflexo (^)), usando-o uma vez para cada etapa na ordem listada acima que você quer ignorar.

Esse exemplo mostra a ordem de qualificação necessária para alcançar uma determinada definição ou parâmetro:

```
ds := DATASET([1], { INTEGER SomeValue });

INTEGER SomeValue := 10; //local definition

myModule(INTEGER SomeValue) := MODULE

  EXPORT anotherFunction(INTEGER SomeValue) := FUNCTION
    tbl := TABLE(ds, {SUM(GROUP, someValue), // 1 - DATASET field
                      SUM(GROUP, ^.someValue), // 84 - FUNCTION parameter
                      SUM(GROUP, ^^someValue), // 42 - MODULE parameter
                      SUM(GROUP, ^^^someValue), // 10 - local definition
                      0});
    RETURN tbl;
  END;

  EXPORT result := anotherFunction(84);
  END;

OUTPUT(myModule(42).result);
```

Nesse exemplo, há quatro ocorrências do nome "SomeValue":

um campo em um DATASET.

uma definição local

um parâmetro de uma estrutura MODULE

um parâmetro de uma estrutura FUNCTION

O código na função TABLE mostra como fazer referência a cada ocorrência separada.

Embora essa sintaxe permita exceções quando necessário, a criação de outra definição com um nome diferente é a melhor solução.

Ações e Definições

Enquanto Definições definem as expressões que podem ser avaliadas, Ações desencadeiam a execução de uma workunit que gera resultados que podem ser vistos. Uma Ação pode avaliar as Definições para gerar seu resultado. Existem várias Ações integradas na ECL (como OUTPUT). Todas as expressões (sem um nome de Definição) são tratadas implicitamente como uma Ação para gerar o resultado da expressão.

Expressões como Ações

Fundamentalmente, qualquer expressão pode ser tratada como uma Ação. Por exemplo,

```
Attr1 := COUNT(Trades);  
Attr2 := MAX(Trades,trd_bal);  
Attr3 := IF (1 = 0, 'A', 'B');
```

são todas definições, mas sem um nome de definição são consideradas apenas como expressões

```
COUNT(Trades);           //execute these expressions as Actions  
MAX(Trades,trd_bal);  
IF (1 = 0, 'A', 'B');
```

que são tratadas como ações e, como tal, podem diretamente gerar valores de resultados se forem enviadas como queries (consultas) para o supercomputador. Basicamente, qualquer expressão da ECL pode ser usada como uma Ação para instigar uma workunit.

Definições como ações

Estas mesmas definições da expressão podem ser executadas ao enviar os nomes das Definições como queries (consultas), conforme este exemplo:

```
Attr1; //These all generate the same result values  
Attr2; // as the previous examples  
Attr3;
```

Ações como Definições

Por outro lado, dar um nome de Definição para qualquer Ação faz com que esta ação se torne uma Definição que, consequentemente, não é mais uma ação diretamente executável. Por exemplo,

```
OUTPUT(Person);
```

é uma ação, porém

```
Attr4 := OUTPUT(Person);
```

é uma definição e não é imediatamente executada ao ser enviada como parte de uma consulta. Para executar a ação integrada à definição, é preciso executar o nome da Definição que foi dado à Ação, como por exemplo:

```
Attr4; // run the previously defined OUTPUT(Person) action
```

Depuração

Essa técnica de execução direta de uma Definição como uma Ação auxilia a depuração de um código ECL complexo. Você pode enviar a Definição como uma consulta para determinar se os valores intermediários foram calculados corretamente antes de avançar para um código mais complexo.

Expressões e Operadores

Expressões e Operadores

Expressions são avaliadas da esquerda para a direita e de dentro para fora (em funções aninhadas). Os Parêntesis podem ser usados para alterar a ordem de precedência padrão da avaliação para todos os operadores.

Operadores Aritméticos

Os operadores padrão aritméticos são suportados para uso em expressões, listados aqui em sua avaliação:

Division	/
Dados Integer	DIV
Modulus Division	%
Multiplication	*
Addition	+
Subtraction	-

Divisão por Zero gera um resultado de valor zero (0), em vez de reportar um erro de "divisão por zero". Isso evita que dados inválidos ou inesperados cancelem um job longo. O comportamento padrão pode ser alterado usando

```
#OPTION ('divideByZero', 'zero'); //evaluate to zero
```

A opção divideByZero pode conter os seguintes valores:

'zero'	Avaliar para 0 - o comportamento padrão.
'fail'	Parar e reportar um erro de divisão por zero.
'nan'	Isto é atualmente suportado apenas para números reais. A Divisão por zero cria um NaN inativo, que será propagado em todas as expressões reais em que é usado. Você pode usar NOT ISVALID(x) para testar se um valor é ou não um NaN. A divisão de inteiros e decimais por zero continua a retornar 0.

Operadores Bitwise

Operadores Bitwise são suportados para uso em expressões, listadas aqui em sua avaliação:

Bitwise AND	&
Bitwise OR	
Bitwise Exclusive OR	^
Bitwise NOT	BNOT

Operadores Bitshift

Operadores Bitshift são suportados para um em expressão "inteiras":

Bitshift Right	>>
Bitshift Left	<<

Operadores de comparação

Os seguintes operadores de comparações são suportados:

Equivalence	=	retorna TRUE ou FALSE
Not Equal	<>	retorna TRUE ou FALSE
Not Equal	!=	retorna TRUE ou FALSE
Less Than	<	retorna TRUE ou FALSE
Greater Than	>	retorna TRUE ou FALSE
Less Than or Equal	<=	retorna TRUE ou FALSE
Greater Than or Equal	>=	retorna TRUE ou FALSE
Equivalence Comparison	<=>	retorna -1, 0, ou 1

O operador “Maior ou igual a” deve primeiramente apresentar o sinal “Maior que” (>). Para a expressão $a <= b$, o operador de comparação de equivalência retorna -1 se $a < b$, 0 se $a = b$, e 1 se $a > b$. Quando as STRINGS são comparadas quanto à equivalência, os espaços atrás são ignorados.

Operadores Lógicos

Os seguintes operadores lógicos são suportados, listados aqui em sua precedência de avaliação:

NOT	Operação Boolean NOT
~	Operação Boolean NOT
AND	Operação Boolean AND
OR	Operação Boolean OR

Agrupando Expressões Lógicas

Quando uma expressão lógica complexa possui múltiplas condições OR, é preciso agrupar as condições OR e ordená-las da menos complexa para a mais complexa a fim de obter um processamento mais eficiente.

Se a probabilidade de ocorrência for conhecida, é preciso ordená-la da mais provável para a menos provável, pois como qualquer parte de uma condição OR composta é avaliada como TRUE, o resto da expressão pode ser ignorado. No entanto, isso não é garantido. Isso também é verdade para a ordem das condições da função MAP.

Sempre que as operações lógicas AND e OR são misturadas em uma mesma expressão, é preciso usar parênteses para agrupar dentro da expressão, garantindo uma avaliação correta e para esclarecer a intenção da expressão. Por exemplo, considere o seguinte:

```
isCurrentRevolv := trades.trd_type = 'R' AND  
                  trades.trd_rate = '0' OR  
                  trades.trd_rate = '1';
```

não gera o resultado pretendido. O uso de parênteses garante a avaliação correta, como mostrado abaixo:

```
isCurrentRevolv := trades.trd_type = 'R' AND  
                  (trades.trd_rate = '0' OR trades.trd_rate = '1');
```

Operador XOR

A função a seguir pode ser usada para realizar uma operação XOR em 2 valores booleanos:

```
BOOLEAN XOR(BOOLEAN cond1, BOOLEAN cond2) :=  
    (cond1 OR cond2) AND NOT (cond1 AND cond2);
```


Operadores de Recordset

Os seguintes operadores de conjunto de registros são suportados (todos exigem que os arquivos foram criados usando estruturas RECORD idênticas):

+	Anexa todos os registros de ambos os arquivos, independentemente da ordem
&	Anexa todos os registros de ambos os arquivos, mantendo a ordem do registro em cada nó
-	Subtrai registros de um arquivo

Exemplo:

```
MyLayout := RECORD
  UNSIGNED Num;
  STRING Number;
END;

FirstRecSet := DATASET([ {1, 'ONE'}, {2, 'Two'}, {3, 'Three'}, {4, 'Four'} ], MyLayout);
SecondRecSet := DATASET([ {5, 'FIVE'}, {6, 'SIX'}, {7, 'SEVEN'}, {8, 'EIGHT'} ], MyLayout);

ExcludeThese := SecondRecSet (Num > 6);

WholeRecSet := FirstRecSet + SecondRecSet;
ResultSet := WholeRecSet - ExcludeThese;

OUTPUT (WholeRecSet);
OUTPUT(ResultSet);
```

Operador Anexar Prefixo.

(+) (*ds_list*) [*options*]

(+)	O prefixo anexo ao operador.
<i>ds_list</i>	Uma lista delimitada por vírgula dos conjuntos de registro a serem anexados (dois ou mais). Todos os conjuntos de registro devem ter estruturas RECORD idênticas.
<i>options</i>	Opcional. Uma lista delimitada por vírgula das opções da lista abaixo.

O operador anexar prefixo (+) oferece mais flexibilidade do que o prefixo simples inserir operadores descrito acima. Ele permite que dicas e outras opções sejam associadas com o operador. Uma sintaxe semelhante será adicionada em uma mudança futura para outros prefixos inserir operadores.

As seguintes *opções* podem ser usadas:

[, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]

UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.

Referência a Linguagem ECL

Expressões e Operadores

<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

Exemplo:

```
ds_1 := (+)(ds1, ds2, UNORDERED);
//equivalent to: ds := ds1 + ds2;

ds_2 := (+)(ds1, ds2);
//equivalent to: ds := ds1 & ds2;

ds_3 := (+)(ds1, ds2, ds3);
//multiple file appends are supported
```

Operadores Set

Os seguintes operadores Set são suportados, listados aqui em sua precedência de avaliação:

+	Anexar (todos os elementos de ambos os conjuntos sem reordenar ou duplicar a remoção do elemento)
---	---

Operadores String

O seguinte operador string é suportado.

+	Concatenação
---	--------------

Operador IN

value **IN** *value_set*

<i>value</i>	O valor a ser localizado no <i>value_set</i> . Trata-se geralmente de um único valor, porém se o <i>value_set</i> for um DICTIONARY com chave de múltiplos componentes, também pode ser uma ROW .
<i>value_set</i>	Um conjunto de valores. Isto pode ser uma expressão do conjunto, a função SET , ou um DICTIONARY .

O operador **IN** é um atalho para uma coleção de condições **OR**. É um operador que buscará uma inclusão no conjunto, resultando em um retorno booleano. O uso de **IN** é muito mais eficiente do que sua expressão equivalente **OR**.

Exemplo:

```
ABCset := ['A', 'B', 'C'];
IsABCStatus := Person.Status IN ABCset;
//This code is directly equivalent to:
// IsABCStatus := Person.Status = 'A' OR
//           Person.Status = 'B' OR
//           Person.Status = 'C';

IsABC(String1 char) := char IN ABCset;
Trades_ABCstat := Trades(IsABC(rate));
// Trades_ABCstat is a record set definition of all those
// trades with a trade status of A, B, or C

//SET function examples
r := {String1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                    {'F'},{'G'},{'H'},{'I'},{'J'}],r);
x := SET(SomeFile(Letter > 'C'),Letter);
y := 'A' IN x; //results in FALSE
z := 'D' IN x; //results in TRUE

//DICTIONARY examples:
rec := {String color,UNSIGNED1 code};
ColorCodes := DATASET([{'Black' ,0 },
                      {'Brown' ,1 },
                      {'Red' ,2 },
                      {'White' ,3 }], rec);

CodeColorDCT := DICTIONARY(ColorCodes,{Code => Color});
OUTPUT(6 IN CodeColorDCT); //false

ColorCodesDCT := DICTIONARY(ColorCodes,{Color,Code});
OUTPUT(ROW({'Red',2},rec) IN ColorCodesDCT);
```

Ver também: Tipos Básicos de Definições, Tipos de definição (Set Definitions), Operadores Lógicos, **PATTERN**, **DICTIONARY**, **ROW**, **SET**, Conjuntos e Filtros, **SET OF**, Operadores Set

Operador BETWEEN

SeekVal **BETWEEN** *LoVal* **AND** *HiVal*

<i>SeekVal</i>	O valor a ser localizado no intervalo inclusivo.
<i>LoVal</i>	O valor baixo no intervalo inclusivo.
<i>HiVal</i>	O valor alto no intervalo inclusivo.

Um operador **BETWEEN** é um atalho para a verificação de intervalo inclusivo usando operadores de comparação padrão ($SeekVal \geq LoVal$ AND $SeekVal \leq HiVal$). $SeekVal \geq LoVal$ AND $SeekVal \leq HiVal$). Pode ser combinado com NOT para inverter a lógica.

Exemplo:

```
X := 10;
Y := 20;
Z := 15;

IsInRange := Z BETWEEN X AND Y;
//This code is directly equivalent to:
// IsInRange := Z >= X AND Z <= Y;

IsNotInRange := Z NOT BETWEEN X AND Y;
//This code is directly equivalent to:
// IsInNotRange := NOT (Z >= X AND Z <= Y);
```

Ver também: Operadores Lógicos, Operadores de comparação

Tipos de valores

Tipos de valores tipos de valores declaram um tipo de atributo quando colocados à esquerda do nome do atributo na definição. Eles também declaram o tipo de parâmetro enviado quando colocados à esquerda do nome do parâmetro na definição. Os Tipos de valores também são convertidos explicitamente de um tipo para outro quando colocados entre parênteses à esquerda da expressão que será convertida.

BOOLEAN

BOOLEAN

Um valor booleano verdadeiro/falso (true ou false). **TRUE** e **FALSE** são palavras-chave reservadas do ECL; são constantes booleanas que podem ser usadas para serem comparadas com um tipo BOOLEAN. Quando BOOLEAN é usado em uma estrutura RECORD , é gerado um valor inteiro de um único byte contendo (1) ou zero (0).

Exemplo:

```
BOOLEAN MyBoolean := SomeAttribute > 10;
    // declares MyBoolean a BOOLEAN Attribute

BOOLEAN MyBoolean(INTEGER p) := p > 10;
    // MyBoolean takes an INTEGER parameter

BOOLEAN Typtrd := trades.trd_type = 'R';
    // Typtrd is a Boolean attribute, likely to be used as a filter
```

Ver também: TRUE/FALSE

INTEGER

[*IntType*] [UNSIGNED] INTEGER[*n*]

[*IntType*] UNSIGNED n

Um valor inteiro de n bytes. Os valores válidos para n são: 1, 2, 3, 4, 5, 6, 7, ou 8. Se n não for especificado para INTEGER, o padrão será 8 bytes.

O *IntType* opcional pode especificar o estilo de valores inteiros ENDIANBIG_ENDIAN (Tipo Sun/UNIX, válido apenas dentro de uma estrutura RECORD) ou (Tipo Intel). Estes dois *IntTypes* possuem ordenações internas de byte opostas. Se a palavra-chave opcional UNSIGNED não estiver presente, o valor inteiro será sinalizado.

Se a palavra-chave opcional UNSIGNED não estiver presente, o valor inteiro será sinalizado. Declarações de valores inteiros não sinalizados podem ser contraídas para UNSIGNED n em vez de UNSIGNED INTEGER n .

Intervalo de Valores INTEGER

Tamanho	Valores sinalizados	Valores não sinalizados
1-byte	-128 a 127	0 a 255
2-byte	-32.768 a 32.767	0 a 65.535
3-byte	-8.388.608 a 8.388.607	0 a 16.777.215
4-byte	-2.147.483.648 a 2.147.483.647	0 a 4.294.967.295
5-byte	-549.755.813.888 a 549.755.813.887	0 a 1.099.511.627.775
6-byte	-140.737.488.355.328 140.737.488.355.327	a 0 a 281.474.976.710.655
7-byte	-36.028.797.018.963.968 36.028.797.018.963.967	a 0 a 72.057.594.037.927.935
8-byte	-9.223.372.036.854.775.808 9.223.372.036.854.775.807	a 0 a 18.446.744.073.709.551.615

Exemplo:

```
INTEGER1 MyValue := MAP(MyString = '1' => MyString, '0');
//MyValue is 1 or 0, changing type from string to integer
UNSIGNED INTEGER1 MyValue := 255; //max value possible in 1 byte
UNSIGNED1 MyValue := 255;
//MyValue contains the max value possible in a single byte
MyRec := RECORD
  LITTLE_ENDIAN INTEGER2 MyLittleEndianValue := 1;
  BIG_ENDIAN INTEGER2 MyBigEndianValue := 1;
  //the physical byte-order is opposite in these two
END
```


REAL

REAL[n]

Um valor de ponto flutuante IEEE padrão de n bytes. Os valores válidos para n são: 4 (valores para 7 dígitos importantes) ou 8 (valores para 15 dígitos importantes). Se n for omitido, REAL será um valor de ponto flutuante de precisão dupla (8 bytes).

Intervalo de Valores REAL

Type Significant Digits Largest Value Smallest Value

Type	Significant Digits	Largest Value	Smallest Value
REAL4	7 (9999999)	3.402823e+038	1.175494e-038
REAL8	15 (999999999999999)	1.797693e+308	2.225074e-308

Exemplo:

```
REAL4 MyValue := MAP(MyString = '1.0' => MyString, '0');  
// MyValue becomes either 1.0 or 0
```

DECIMAL

[UNSIGNED] DECIMAL_n [_y]

UDECIMAL_n [_y]

Um valor decimal de n dígitos totais. Se o valor de `_y` está presente, o `y` define o número de casas decimais no valor. Pode haver no máximo 32 dígitos inteiros e 32 dígitos fracionários.

Se a palavra-chave `UNSIGNED` for omitida, o nibble mais à direita deterá o sinal. As declarações decimais não sinalizadas podem ser compactadas para usar a sintaxe opcional `UDECIMAL n` em vez de `UNSIGNEDDECIMAL n` .

O uso exclusivo de valores DECIMAIS DECIMAL em computação invoca as bibliotecas matemáticas Binary Coded Decimal (BCD) (matemática de base 10), permitindo até 32 dígitos de precisão (que pode estar em qualquer um dos lados do ponto decimal).

Exemplo:

```
DECIMAL5_2 MyDecimal := 123.45;
    //five total digits with two decimal places

OutputFormat199 := RECORD
    UNSIGNED DECIMAL9 Person.SSN;
    //unsigned packed decimal containing 9 digits,
    // occupying 5 bytes in a flat file

UDECIMAL10 Person.phone;
    //unsigned packed decimal containing 10 digits,
    // occupying 5 bytes in a flat file

END;
```

STRING

[*StringType*] **STRING**[*n*]

Uma string de caracteres de *n* bytes, completada com espaços (não terminada por nulo). If *n* se *n* for omitido, a string terá o tamanho variável necessário para conter o resultado do parâmetro convertido ou passado. Você pode usar indexação de conjunto de qualquer string para extrair uma substring..

O *StringType* opcional pode especificar ASCII ou EBCDIC. Se *StringType* não for encontrado, os dados estarão no formato ASCII. A definição de um atributo EBCDIC STRING como valor constante de string implica em uma conversão de ASCII para EBCDIC. No entanto, a definição de um atributo EBCDIC STRING como um valor de constante de string hexadecimal não implica em nenhuma conversão, pois supõe-se que o programador tenha fornecido o valor EBCDIC hexadecimal correto.

O limite de tamanho máximo para qualquer valor DATA é 4GB.

Exemplo:

```
STRING1 MyString := IF(SomeAttribute > 10,'1','0');  
    // declares MyString a 1-byte ASCII string  
  
EBCDIC STRING3 MyString1 := 'ABC';  
    //implicit ASCII to EBCDIC conversion  
  
EBCDIC STRING3 MyString2 := x'616263';  
    //NO conversion here
```

Ver também: LENGTH, TRIM, Classificação e indexação de conjuntos, Hexadecimal String

QSTRING

QSTRING[*n*]

Uma variação de dados compactados da **STRING** que usa apenas 6 bits por caractere para diminuir os requisitos de armazenagem de strings maiores. O conjunto de caracteres é limitado às letras A-Z, aos números 0-9, ao espaço em branco, e ao seguinte conjunto de caracteres especiais:

```
! " # $ % & ' ( ) * + , - . / ; < = > ? @ [ \ ] ^ _
```

Se *n* for omitido, a **QSTRING** terá comprimento variável para o tamanho necessário para conter o resultado de um parâmetro de conversão ou especificado. Você pode usar indexação de conjunto de qualquer string para extrair uma substring.

O limite de tamanho máximo para qualquer valor **DATA** é 4GB.

Exemplo:

```
QSTRING12 CompanyName := 'LEXISNEXIS';  
// uses only 9 bytes of storage instead of 12
```

Ver também: **STRING**, **LENGTH**, **TRIM**, Classificação e indexação de conjuntos.

UNICODE

UNICODE[_*locale*][*n*]

Uma string de caracteres com codificação UTF-16 com *n* caracteres, complementada por espaços da mesma forma que **STRING**. Se *n* for omitido, a string terá o tamanho variável necessário para conter o resultado do parâmetro convertido ou passado. *locale* é opcional e especifica um código de local Unicode válido, como especificado nos padrões ISO 639 e 3166 (não é necessário se **LOCALE** é especificado na estrutura **RECORD** que contém a definição do campo).

A conversão do tipo **UNICODE** para **VARUNICODE**, **STRING** ou **DATA** é permitida. A conversão para qualquer outro tipo implicará antes em uma conversão implícita para **STRING** e depois para o tipo de valor pretendido.

O limite de tamanho máximo para qualquer valor **UNICODE** é 4GB.

Exemplo:

```
UNICODE16 MyUNIStrIng := U'1234567890ABCDEF';  
    // utf-16-encoded string  
UNICODE4 MyUnicodeString := U'abcd';  
    // same as: (UNICODE)'abcd'  
UNICODE_de5 MyUnicodeString := U'abcd\353';  
    // becomes 'abcdë' with a German locale  
UNICODE_de5 MyUnicodeString := U'abcdë';  
    // same as previous example
```

UTF8

UTF8[_locale][_n]

Uma string de caracteres Unicode codificada em UTF-8 de n caracteres, preenchida com espaço da mesma forma que STRING. Se $_n$ for omitido, a string terá comprimento variável até o tamanho necessário para conter o resultado da conversão ou do parâmetro passado. O *locale* opcional especifica um código de localidade Unicode válido, conforme especificado nos padrões ISO 639 e 3166 (não necessário se LOCALE for especificado na estrutura RECORD que contém a definição do campo).

A conversão do tipo UTF8 para UNICODE, VARUNICODE, STRING ou DATA é permitida. A conversão para qualquer outro tipo implicará antes em uma conversão implícita para STRING e depois para o tipo de valor pretendido.

O limite de tamanho máximo para qualquer valor UTF8 é 4GB.

Exemplo:

```
UTF8 FirstName := U8'Noël';           // utf-8-encoded string
UTF8_de MyUnicodeString := U8'abcd\353'; // becomes 'abcdë' with a German locale
UTF8_4 FirstName4 := U8'Noël';         // 4-character utf-8-encoded string
UTF8_de_5 MyUnicodeString5 := U8'abcd\353'; // becomes 'abcdë' with a German locale
```

DATA

DATA[*n*]

Um bloco de dados "" packed hexadecimal de *n* bytes, preenchido com zeros (e não com espaços). Se *n* for omitido, a string terá o tamanho variável necessário para conter o resultado do parâmetro convertido ou passado. A conversão de tipo é permitida apenas para uma STRING ou UNICODE que possui o mesmo número de bytes.

Este tipo é especialmente útil para os dados que contêm BLOB (Binary Large Object). Consulte o artigo do **Guia do Programador Trabalhando com BLOBs** para obter mais informações sobre este assunto.

O limite de tamanho máximo para qualquer valor DATA é 4GB.

Exemplo:

```
DATA8 MyHexString := x'1234567890ABCDEF';  
    // an 8-byte data block - hex values 12 34 56 78 90 AB CD EF
```

VARSTRING

VARSTRING $[n]$

Uma string de caracteres terminada por nulo que contém n bytes de dados. Se n for omitido, a string terá o tamanho variável necessário para conter o resultado do parâmetro convertido ou passado. Você pode usar indexação de conjunto de qualquer string para extrair uma substring.

O limite de tamanho superior para qualquer valor de VARSTRING é de 4 GB.

Exemplo:

```
VARSTRING3 MyString := 'ABC';  
    // declares MyString a 3-byte null-terminated string
```

Ver também: LENGTH, TRIM, Classificação e indexação de conjuntos

VARUNICODE

VARUNICODE[*locale*][*n*]

Uma string de caracteres Unicode com codificação UTF-16 de *n* caracteres, terminada por nulo (não complementada com espaços). A *n* pode ser omitido apenas quando usado como um tipo de parâmetro. *locale* é opcional e especifica um código de local Unicode válido, como especificado nos padrões ISO 639 e 3166 (não é necessário se LOCALE é especificado na estrutura RECORD que contém a definição do campo).

A conversão do tipo UNICODE para VARUNICODE, STRING ou DATA é permitida. A conversão para qualquer outro tipo implicará antes em uma conversão implícita para STRING e depois para o tipo de valor pretendido.

O limite de tamanho superior para qualquer valor de VARUNICODE é de 4 GB.

Exemplo:

```
VARUNICODE16 MyUNIStrIng := U'1234567890ABCDEF';  
    // utf-16-encoded string  
VARUNICODE4 MyUnicodeString := U'abcd';  
    // same as: (UNICODE)'abcd'  
VARUNICODE5 MyUnicodeString := U'abcd\353';  
    // becomes 'abcdë'  
VARUNICODE5 MyUnicodeString := U'abcdë';  
    // same as previous example
```

SET OF

SET [OF *type*]

<i>type</i>	O tipo de valor dos dados no conjunto. Os tipos de valor válidos são: INTEGER, REAL, BOOLEAN, STRING, UNICODE, DATA, or DATASET(<i>recstruct</i>). Se omitido, <i>type</i> é INTEGER.
-------------	---

O tipo **SET OF** define atributos que são um conjunto de elementos de dados. Todos os elementos do conjunto devem ser do mesmo *value type*. O valor padrão poderá ser a palavra-chave ALL se o tipo de valor for SET (consulte a palavra-chave SET) para indicar todos os valores possíveis para esse tipo de conjunto, ou colchetes vazios ([]) para indicar que não há nenhum valor possível para esse tipo de conjunto.

Exemplo:

```
SET OF INTEGER1 SetIntOnes := [1,2,3,4,5];
SET OF STRING1 SetStrOnes := ['1','2','3','4','5'];
SET OF STRING1 SetStrOne1 := (SET OF STRING1)SetIntOnes;
//type casting sets is allowed
r := {STRING F1, STRING2 F2};
SET OF DATASET(r) SetDS := [ds1, ds2, ds3];

StringSetFunc(SET OF STRING passedset) := AstringValue IN passedset;
//a set of string constants will be passed to this function
HasNarCode(SET s) := Trades.trd_narr1 IN s OR Trades.trd_narr2 IN s;
// HasNarCode takes a parameter that specifies the set of valid
// Narrative Code values (all INTEGERS)
SET OF INTEGER1 SetClsdNar := [65,66,90,114,115,123];
NarCodeTrades := Trades(HasNarCode(SetClsdNar));
// Using HasNarCode(SetClsdNar) is equivalent to:
// Trades.trd_narr1 IN [65,66,90,114,115,123] OR
// Trades.trd_narr2 IN [65,66,90,114,115,123]
```

Ver também: Funções dos atributos (Especificações de parâmetros), Classificação e indexação de conjuntos

TYPEOF

TYPEOF(*expression*)

<i>expression</i>	Uma expressão que define o tipo de valor. Pode ser o nome de um campo de dados, de um parâmetro especificado, de uma função ou de um atributo que fornece o tipo de valor (incluindo estruturas RECORD). Esta deve ser uma expressão legal para o escopo atual, mas não é avaliada quanto ao seu valor.
-------------------	---

A declaração **TYPEOF** permite definir um atributo ou parâmetro cujo tipo de valor é “o mesmo” da *expressão*. Isto é válido para ser usado em qualquer lugar onde um tipo de valor explícito seja válido.

Seu uso mais comum seria para especificar o tipo de retorno de uma função TRANSFORM como "o mesmo" da estrutura de um dataset ou de um recorset.

Exemplo:

```
STRING3 Fred := 'ABC'; //declare Fred as a 3-byte string
TYPEOF(Fred) Sue := Fred; //declare Sue as "just like" Fred
```

Ver também: Estrutura TRANSFORM

RECORDOF

RECORDOF(*recordset* , [LOOKUP])

<i>recordset</i>	O conjunto de registros de dados cuja estrutura RECORD será usada. Pode ser um DATASET ou qualquer conjunto de registros derivado. Se o atributo LOOKUP for usado, isso pode ser um nome de arquivo.
LOOKUP	Opcional. Especifica que o layout de arquivo deve ser consultado no tempo de compilação. Consulte <i>Resolução de layout de arquivo no tempo de compilação no Guia do Programador</i> para obter mais detalhes.

A declaração **RECORDOF** especifica o uso de apenas o layout de registro do *recordset* em situações onde seria necessário herdar a estrutura dos campos, mas não seus valores padrão, como declarações de DATASET filho dentro de estruturas RECORD.

Essa função permite que você mantenha as estruturas RECORD locais no DATASET, cujo layout são definidos por essas estruturas, e ainda consiga referenciar a estrutura (sem valores padrão) onde necessário.

Exemplo:

```
Layout_People_Slim := RECORD
  STD_People.RecID;
  STD_People.ID;
  STD_People.FirstName;
  STD_People.LastName;
  STD_People.MiddleName;
  STD_People.NameSuffix;
  STD_People.FileDate;
  STD_People.BureauCode;
  STD_People.Gender;
  STD_People.BirthDate;
  STD_People.StreetAddress;
  UNSIGNED8 CSZ_ID;
END;

STD_Accounts := TABLE(UID_Accounts,Layout_STD_AcctsFile);

CombinedRec := RECORD,MAXLENGTH(100000)
  Layout_People_Slim;
  UNSIGNED1 ChildCount;
  DATASET(RECORDOF(STD_Accounts)) ChildAccts;
END;

//This ChildAccts definition is equivalent to:
// DATASET(Layout_STD_AcctsFile) ChildAccts;
//but doesn't require Layout_STD_AcctsFile to be visible (SHARED or
// EXPORT)
```

Ver também: DATASET, Estrutura RECORD

ENUM

ENUM([*type* ,] *name* [=value] [, *name* [=value] ...])

<i>type</i>	O tipo de valor numérico dos <i>valores</i> . Se omitido, o padrão para UNSIGNED4.
<i>name</i>	O rótulo do <i>valor</i> enumerado.
<i>value</i>	O valor numérico a ser associado ao <i>nome</i> . Se omitido, o <i>valor</i> será o <i>valor</i> anterior mais um (1). Se todos os <i>valores</i> forem omitidos, a enumeração começa com o número um (1).

A declaração **ENUM** especifica os valores constantes para facilitar a leitura do código.

Exemplo:

```
GenderEnum := ENUM(UNSIGNED1, Male, Female, Either, Unknown);
//values are 1, 2, 3, 4

Pflg := ENUM(None=0, Dead=1, Foreign=2, Terrorist=4, Wanted=Terrorist*2);
//values are 0, 1, 2, 4, 8

namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    GenderEnum gender;
    INTEGER2 age := 25;
END;

namesTable2 := DATASET([{'Foreman', 'George', GenderEnum.Male, Pflg.Foreign},
                        {'Bin', 'O', GenderEnum.Male, Pflg.Foreign+Pflg.Terrorist+Pflg.Wanted}
                        ], namesRecord);

OUTPUT(namesTable2);

myModule(UNSIGNED4 baseError, STRING x) := MODULE
    EXPORT ErrorCode := ENUM( ErrorBase = baseError,
                             ErrNoActiveTable,
                             ErrNoActiveSystem,
                             ErrFatal,
                             ErrLast);
    EXPORT reportX := FAIL(ErrorCode.ErrNoActiveTable, 'No ActiveTable in ' + x);
END;

myModule(100, 'Call1').reportX;
myModule(300, 'Call2').reportX;
```

Conversão de Conversão

Conversão Explícita

O uso mais comum de tipos de valor é converter explicitamente de um tipo para outro em expressões. Para isso, basta colocar o tipo de valor pretendido entre parênteses. Isso cria um operador de conversão. Em seguida, coloque o operador de conversão imediatamente à esquerda da expressão a ser convertida.

Os dados serão convertidos do formato original para o novo formato (para manter o mesmo padrão de bits, consulte a função incorporada **TRANSFER**).

```
MyBoolean := (BOOLEAN) IF(SomeAttribute > 10,1,0);  
           // casts the INTEGER values 1 and 0 to a BOOLEAN TRUE or FALSE  
MyString := (STRING1) IF(SomeAttribute > 10,1,0);  
           // casts the INTEGER values 1 and 0 to a 1-character string  
           // containing '1' or '0'  
MyValue := (INTEGER) MAP(MyString = '1' => MyString, '0');  
           // casts the STRING values '1' and '0' to an INTEGER 1 or 0  
MySet := (SET OF INTEGER1) [1,2,3,4,5,6,7,8,9,10];  
           //casts from a SET OF INTEGER8 (the default) to SET OF INTEGER1
```

Conversão Implícita

Durante a avaliação da expressão, podem ocorrer conversões implícitas para diversos tipos de valores para avaliar corretamente a expressão. A conversão implícita significa sempre a promoção de um tipo de valor para outro: INTEGER para STRING ou INTEGER para REAL. Os tipos BOOLEAN não podem ser envolvidos em expressões de modo misto. Por exemplo, ao avaliar uma expressão usando valores INTEGER e REAL, o INTEGER será promovido para REAL no momento em que os dois valores são misturados e o resultado será um valor REAL.

INTEGER e REAL podem ser misturados livremente nas expressões. No ponto de contato entre eles, a expressão será tratada como REAL. Até esse ponto de contato, a expressão poderá ser avaliada com o comprimento INTEGER. A divisão de valores INTEGER promove implicitamente os dois operandos para REAL antes da divisão.

Esta expressão: $(1+2+3+4)*(1.0*5)$

é avaliada como: $(REAL)((INTEGER)1+(INTEGER)2+(INTEGER)3+(INTEGER)4)*(1.0*(REAL)5)$

e: $5/2+4+5$ é avaliada como: $(REAL)5/(REAL)2+(REAL)4+(REAL)5$

ao passo que: $'5' + 4$ é avaliado como: $5 + (STRING)4$ //concatenação

Os operadores de comparação são tratados como qualquer outra expressão de modo misto. Funções incorporadas que recebem vários valores e que podem retornar um ou mais desses valores (como MAP ou IF) são tratadas como expressões de modo misto e retornam o tipo básico comum. Esse tipo comum deve ser acessível às conversões implícitas padrão.

Transferência de Tipo

A conversão de tipos converte dados do seu formato original para o novo formato. Para manter o mesmo padrão de bits, você deve usar a função incorporada **TRANSFER** ou a sintaxe de transferência de tipo, que é semelhante à sintaxe de conversão de tipos, com a adição de colchetes angulares (*>valuetype<*).

```
INTEGER1 MyInt := 65; //MyInt is an integer value 65  
STRING1 MyVal := (>STRING1<) MyInt; //MyVal is "A" (ASCII 65)
```

Regras de Conversão

De	Para	Resulta em
INTEGER	STRING	Representação ASCII ou EBCDIC do valor
DECIMAL	STRING	Representação ASCII ou EBCDIC do valor, incluindo casas decimais e sinal
REAL	STRING	Representação ASCII ou EBCDIC do valor, incluindo casas decimais e sinal; pode ser expressa em notação científica
UNICODE	STRING	Representação ASCII ou EBCDIC com quaisquer caracteres não existentes aparecendo como código de controle SUBstitute (0x1A em ASCII ou 0x3F em EBCDIC) e quaisquer caracteres ASCII ou EBCDIC inválidos aparecendo como o ponto de código de substituição (0xFFFD)
UTF8	STRING	Representação ASCII ou EBCDIC com quaisquer caracteres não existentes aparecendo como código de controle SUBstitute (0x1A em ASCII ou 0x3F em EBCDIC) e quaisquer caracteres ASCII ou EBCDIC inválidos aparecendo como o ponto de código de substituição (0xFFFD)
STRING	QSTRING	Representação ASCII em maiúsculas
INTEGER	UNICODE	Representação UNICODE do valor
DECIMAL	UNICODE	Representação UNICODE do valor, incluindo casas decimais e sinal
REAL	UNICODE	Representação UNICODE do valor, incluindo casas decimais e sinal; pode ser expressa em notação científica
INTEGER	UTF8	Representação UTF8 do valor
DECIMAL	UTF8	Representação UNICODE do valor, incluindo casas decimais e sinal
REAL	UTF8	Representação UTF8 do valor, incluindo casas decimais e sinal; pode ser expressa em notação científica
INTEGER	REAL	O valor será convertido sem perda de precisão quando o valor for maior que 15 dígitos significativos
INTEGER	REAL4	O valor será convertido sem perda de precisão quando o valor for maior que 7 dígitos significativos
STRING	REAL	Parte de sinal, inteiro e casas decimais do valor da string
DECIMAL	REAL	O valor será convertido sem perda de precisão quando o valor for maior que 15 dígitos significativos
DECIMAL	REAL4	O valor será convertido sem perda de precisão quando o valor for maior que 7 dígitos significativos
INTEGER	DECIMAL	Perda de precisão se DECIMAL for muito pequeno
REAL	DECIMAL	Perda de precisão se DECIMAL for muito pequeno
STRING	DECIMAL	Parte de sinal, inteiro e casas decimais do valor da string
STRING	INTEGER	Parte de sinal e inteiro do valor da string
REAL	INTEGER	Somente valor inteiro, a parte decimal é truncada
DECIMAL	INTEGER	Somente valor inteiro, a parte decimal é truncada
INTEGER	BOOLEAN	0 = FALSE, qualquer outro valor = TRUE
BOOLEAN	INTEGER	FALSE = 0, TRUE = 1

Referência a Linguagem ECL
Tipos de valores

STRING	BOOLEAN	“ = FALSE, qualquer outro valor = TRUE
BOOLEAN	STRING	FALSE = ", TRUE = 'I'
DATA	STRING	O valor é convertido sem tradução
STRING	DATA	O valor é convertido sem tradução
DATA	UNICODE	O valor é convertido sem tradução
UNICODE	DATA	O valor é convertido sem tradução
DATA	UTF8	O valor é convertido sem tradução
UTF8	DATA	O valor é convertido sem tradução
UTF8	UNICODE	O valor é convertido sem tradução
UNICODE	UTF8	O valor é convertido sem tradução

As regras de conversão de STRING de e para qualquer outro tipo numérico também se aplicam igualmente a todos os tipos de string. Todas as regras de conversão se aplicam igualmente a conjuntos (usando a sintaxe SET OF *type*).

Estruturas de registros e arquivos

Estrutura RECORD

attr := **RECORD** [(*baserec*)] [, **MAXLENGTH**(*length*)] [, **LOCALE**(*locale*)] [, **PACKED**]

fields ;

[**IFBLOCK**(*condition*)

fields ;

END;]

[=> *payload*]

END;

<i>attr</i>	O nome da estrutura RECORD a ser usada posteriormente em outras definições.
<i>baserec</i>	Opcional. O nome de uma estrutura RECORD da qual todos os campos serão herdados. Qualquer estrutura RECORD que herdar os campos <i>baserec</i> desta maneira, se tornará compatível com qualquer função TRANSFORM definida a adotar um parâmetro do tipo <i>baserec</i> (os campos <i>adicionais</i> serão, obviamente, perdidos).
MAXLENGTH	Opcional. Esta opção é usada para criar índices que são compatíveis com versões anteriores às versões 3.0. Especifica o número máximo de caracteres permitidos na estrutura RECORD ou campo. MAXLENGTH na estrutura RECORD substitui qualquer MAXLENGTH em uma definição de campo, a qual substitui qualquer MAXLENGTH especificado na estrutura TYPE se o <i>datatype</i> nomear um tipo de dados de natureza diferente. Esta opção define o tamanho máximo dos registros de comprimento variável. Se omitida, os registros de tamanho fixo utilizarão o tamanho mínimo exigido e os registros de comprimento variável gerarão um aviso. O tamanho máximo padrão de um registro contendo campos de comprimento variável é de 4.096 bytes (isso pode ser substituído usando <i>#OPTION(maxLength,####)</i> para alterar o valor padrão). O tamanho máximo do registro deve ser definido de forma mais conservadora possível, sendo melhor definir em termos “por campo” (consulte a seção Modificadores de filtro abaixo).
<i>length</i>	Uma constante de valor inteiro que especifica o número máximo de caracteres permitido.
LOCALE	Opcional. Especifica a <i>localidade</i> Unicode para quaisquer campos UNICODE.
<i>locale</i>	Uma constante de string que contém um código de localidade válido, como especificado nas normas ISO 639 e 3166.
PACKED	Opcional. Especifica a ordem para qual os <i>campos</i> devem ser mudados a fim de melhorar a eficiência (como p.ex., mudar os campos de comprimento variável após os campos de comprimento fixo).
<i>fields</i>	Declarações de campo. Consulte abaixo quanto as sintaxes apropriadas.
IFBLOCK	Opcional. Um bloco de <i>campos</i> que recebe dados "em tempo real" apenas se a <i>condição</i> for atendida. O IFBLOCK deve terminar com um END . Isto é usado para definir os registros de comprimento variável. Se a expressão da <i>condição</i> referenciar os <i>campos</i> no RECORD que precede o IFBLOCK, essas referências devem usar SELF. antes do nome do campo para desambiguar a referência.

Referência a Linguagem ECL
Estruturas de registros e arquivos

<i>condition</i>	Uma expressão lógica que define quando os <i>campos</i> no IFBLOCK receberão dados "em tempo real". Se a expressão não for "true" (verdadeira), os <i>campos</i> conterão seus valores padrão declarados. Se não houver valores padrão, os <i>campos</i> ficarão em branco ou exibirão o número zero.
=>	Opcional. O delimitador entre a lista dos <i>campos</i> chave e a <i>carga útil (payload)</i> quando a estrutura RECORD for usada pela declaração DICTIONARY. Normalmente, esta é uma estrutura embutida que usa as chaves ({ }) em vez de RECORD e END.
<i>payload</i>	A lista dos campos sem <i>chave</i> no DICTIONARY.

Os layouts de registro são definições cuja expressão é uma estrutura RECORD terminada pela palavra-chave END. O nome *attr* cria um tipo de valor definido pelo usuário que pode ser usado em funções embutidas e em definições da função TRANSFORM. O delimitador entre as definições de campo em uma estrutura RECORD pode ser na forma de ponto e vírgula (;) ou apenas vírgula (,).

Definições de Registro em Linha

As chaves ({}) são equivalentes léxicos às palavras-chave RECORD e END que podem ser usadas em qualquer lugar onde RECORD e END são adequados. Ambas as formas (RECORD/END ou {}) podem ser usadas para criar formatos de registro "dinâmicos" nas funções que exigem estruturas de registro (OUTPUT, TABLE, DATASET etc.), em vez de definir o registro como uma definição separada.

Definições de Campo

Todas as declarações de campo em uma estrutura RECORD devem usar uma das seguintes sintaxes:

	<i>datatype identifier [{modifier}] [:= defaultvalue] ;</i>
	<i>identifier := defaultvalue ;</i>
	<i>defaultvalue ;</i>
	<i>sourcefield ;</i>
	<i>reconstruct [identifier] ;</i>
	<i>sourcedataset ;</i>
	<i>childdataset identifier [{ modifier }] ;</i>

<i>datatype</i>	O tipo de valor do campo de dados. Pode ser um dataset filho (consultar DATASET). Se omitido, o tipo de valor corresponde ao tipo de resultado da expressão <i>defaultvalue</i> .
<i>identifier</i>	O nome do campo. Se omitido, o padrão é: <i>defaultvalue</i> se omitido, a expressão <i>defaultvalue</i> define uma coluna sem nome que poderá não ser referenciada em ECL subsequente.
<i>defaultvalue</i>	Opcional. Uma expressão que define a origem dos dados (para operações que exigem uma origem de dados, tais como TABLE e PARSE). Pode ser uma constante, uma expressão ou uma definição que fornece o valor.
<i>modifier</i>	Opcional. Uma das palavras-chave listadas na seção Modificadores de filtro abaixo.
<i>sourcefield</i>	Um campo de dados previamente definido que fornece de forma implícita o <i>datatype</i> (tipo de dados), <i>identifier</i> (identificador), e o <i>defaultvalue</i> (valor padrão) do novo campo – herdado do <i>sourcefield</i> (campo de origem).
<i>reconstruct</i>	Uma estrutura RECORD previamente definida. Consulte a seção Herança de campo abaixo.

Referência a Linguagem ECL
Estruturas de registros e arquivos

<i>sourcedataset</i>	Um DATASET previamente definido ou uma definição do recordset derivado. Consulte a seção Herança de campo abaixo.
<i>childdataset</i>	Uma declaração de dataset filho (veja as discussões DATASET e DICTIONARY), que implicitamente define todos os campos do filho no <i>datatype</i> , <i>identifier</i> (identificador), e <i>defaultvalue</i> (valor padrão) já definidos (se estiver presente na estrutura RECORD do dataset secundário).

As definições de campo sempre devem definir o *datatype* e o *identifier* (identificador) de cada campo, seja de forma implícita ou explícita. Se a estrutura RECORD for usada por TABLE, PARSE, ROW, ou por qualquer outra função que cria um recordset de resultado, o *defaultvalue* (valor padrão) também deve estar definido de forma implícita ou explícita em cada campo. Nos casos em que um campo é definido no estilo de um campo no dataset já em escopo, o *identifier* (identificador) deve ter um nome que já está sendo usado no dataset em escopo, contanto que o *datatype* seja definido de forma explícita.

Herança de Campo

As definições de campo podem ser herdadas de uma estrutura RECORD ou de um DATASET previamente definidos. Quando uma *recstruct* (uma estrutura RECORD) é especificada para herdar os campos, os novos campos são definidos de forma implícita através do uso do *datatype* e do *identifier* (identificador) de todas as definições de campo existentes na *recstruct*. Quando um *sourcedataset* (um DATASET previamente definido ou uma definição de recordset) é especificado para herdar os campos, os novos campos são definidos de forma implícita através do uso do *datatype* (tipo de dados), *identifier*, e do *defaultvalue* (valor padrão) de todos os campos (tornando-o utilizável pelas operações que exigem origem de dados, tais como TABLE e PARSE). Opcionalmente, ambas estas formas podem ter seu próprio *identifier* (identificador), permitindo referenciar todo o conjunto de campos herdados como uma única entidade.

Também é possível usar operadores lógicos (AND, OR, e NOT) para incluir/excluir determinados campos da herança, como descrito aqui:

<i>R1 AND R2</i>	Intersecção	Todos os campos declarados em R1 e R2 <i>R1 and R2</i>
<i>R1 OR R2</i>	União	Todos os campos declarados em R1 ou R2 <i>R1 or R2</i>
<i>R1 AND NOT R2</i>	Diferença	Todos os campos em <i>R1</i> que não estão presentes no <i>R2</i>
<i>R1 AND NOT F1</i>	Exceção	Todos os campos em <i>R1</i> , exceto o campo especificado (<i>F1</i>)
<i>R1 AND NOT [(F1, F2)]</i>	Exceção	Todos os campos em <i>R1</i> , exceto os listados em colchetes (<i>F1</i> e <i>F2</i>)

O sinal de subtração (-) é um sinônimo de AND NOT; portanto, *R1-R2* é equivalente a *R1 AND NOT R2*.

Será considerado um erro se os registros contiverem os mesmos nomes de campo cujos tipos de valores não são correspondentes, ou se você não tiver nenhum campo (como p.ex.: A-A). É preciso se certificar de que qualquer MAXLENGTH/MAXCOUNT esteja especificado corretamente em cada campo em ambas as estruturas RECORD.

Exemplo:

```
R1 := {STRING1 F1,STRING1 F2,STRING1 F3,STRING1 F4,STRING1 F5};
R2 := {STRING1 F4,STRING1 F5,STRING1 F6};
R3 := {R1 AND R2}; //Intersection - fields F4 and F5 only
R4 := {R1 OR R2}; //Union - all fields F1 - F6
R5 := {R1 AND NOT R2}; //Difference - fields F1 - F3
R6 := {R1 AND NOT F1}; //Exception - fields F2 - F5
R7 := {R1 AND NOT [F1,F2]}; //Exception - fields F3 - F5

//the following two RECORD structures are equivalent:
```

```
C := RECORD, MAXLENGTH(x)
  R1 OR R2;
END;
```

```
D := RECORD, MAXLENGTH(x)
  R1;
  R2 AND NOT R1;
END;
```

Modificadores de Campo

A seguinte lista de modificadores de campo está disponível para uso nas definições de campo:

	{ MAXLENGTH (<i>length</i>) }
	{ MAXCOUNT (<i>records</i>) }
	{ XPATH ('tag') }
	{ XMLDEFAULT ('value') }
	{ DEFAULT (<i>value</i>) }
	{ VIRTUAL (<i>fileposition</i>) }
	{ VIRTUAL (<i>localfileposition</i>) }
	{ VIRTUAL (<i>logicalfilename</i>) }
	{ BLOB }

{ MAXLENGTH (<i>length</i>) }	Especifica o número máximo de caracteres permitidos no campo (consulte a opção MAXLENGTH acima).
{ MAXCOUNT (<i>records</i>) }	Especifica o número máximo de <i>registros</i> permitidos em um campo do DATASET filho (semelhante ao MAXLENGTH acima).
{ XPATH ('tag') }	Especifica a <i>tag</i> XML ou JSON que contém os dados, em uma estrutura RECORD que define dados XML ou JSON. Isso substitui o nome da <i>tag</i> padrão (o campo em caixa baixa <i>identifier</i>). Consulte a seção Suporte XPATH abaixo para obter detalhes.
{ XMLDEFAULT ('value') }	Especifica o <i>valor</i> padrão para o campo. O <i>valor</i> deve ser constante.
{ DEFAULT (<i>value</i>) }	Especifica o <i>valor</i> padrão para o campo. O <i>valor</i> deve ser constante. Este <i>valor</i> será usado: <ol style="list-style-type: none"> Quando a busca DICTIONARY não retornar nenhuma correspondência. Quando um registro fora do intervalo for buscado usando ds[n] (como em ds[5], quando ds contém apenas 4 registros). Nos registros padrão especificados para as funções TRANSFORM em JUNÇÕES NÃO INTERNAS (non-INNER JOINS) onde não há uma linha correspondente. Ao selecionar automaticamente os valores de campo no TRANSFORM usando SELF = [].
{ VIRTUAL (<i>fileposition</i>) }	Especifica que o campo é um campo VIRTUAL contendo a posição de byte relativo do registro dentro do arquivo inteiro (o

Referência a Linguagem ECL
Estruturas de registros e arquivos

	ponteiro do registro). Isto deve ser um campo UNSIGNED8 e também o último campo, uma vez que ele realmente existe apenas quando o arquivo é carregado do disco para a memória (daí o nome "virtual").
{ VIRTUAL(localfileposition) }	Especifica a posição local do byte dentro de uma parte do arquivo distribuído em um único nó: o primeiro bit é um conjunto, os próximos 15 bits especificam o número da parte e os últimos 48 bits especificam a posição relativa do byte dentro da parte. Isto deve ser um campo UNSIGNED8 e também o último campo, uma vez que ele realmente existe apenas quando o arquivo é carregado do disco para a memória (daí o nome "virtual").
{ VIRTUAL(logicalfilename) }	Especifica o nome do arquivo lógico do arquivo distribuído. Isso deve ser um campo de STRING. Se a leitura estiver sendo feita a partir de um superarquivo, o valor será o arquivo lógico atual no superarquivo.
{ BLOB }	Especifica que o campo é armazenado separadamente da entrada do nó folha no INDEX. Isto é aplicado especificamente aos campos na carga útil de um INDEX para permitir mais de 32K de dados por entrada de índice. Os dados BLOB são armazenados dentro do arquivo do índice, mas não com o restante do registro. O acesso aos dados BLOB exige uma busca adicional.

Suporte a XPATH

XPATH support é um subconjunto limitado da especificação XPATH completa, basicamente especificado como:

node[qualifier] / node[qualifier] ...

<i>node</i>	Pode conter elementos curinga.
<i>qualifier</i>	Pode ser um nó ou um atributo; ou uma expressão única e simples de igualdade, de desigualdade, ou de comparações numéricas ou alfanuméricas, ou valores do índice do nó. Nenhuma função ou aritmética embutida, etc., são suportadas. A comparação da string é indicada quando o lado direito da expressão estiver entre aspas.

Estes operadores são válidos para comparações:

<, <=, >, >=, =, !=

Um exemplo de xpath suportado:

```
/a/*/c*/*d/e[@attr]/f[child]/g[@attr="x"]/h[child>="5"]/i[@x!="2"]/j
```

É possível emular as condições AND da seguinte forma:

```
/a/b[@x="1"][@y="2"]
```

Adicionalmente, há uma conversão não padronizada de XPATH para extrair o texto de uma correspondência usando colchetes angulares vazios (<>):

```
R := RECORD
STRING blah{xpath('a/b<>')};
//contains all of b, including any child definitions and values
END;
```

Um XPATH de um valor não pode ser ambíguo. Se o elemento ocorrer diversas vezes, é preciso usar a operação ordinal (por exemplo, /foo[1]/bar) para selecionar explicitamente a primeira ocorrência.

Para XML ou JSON DATASETS lendo e processando os resultados do função SOAPCALL, a seguinte sintaxe XPATH é especificamente suportada:

1) Para campos de valor scalar simples, se houver um XPATH especificado ele será usado; caso contrário, será usado o *identifíer* (*identificador*) de caixa baixa do campo.

```
STRING name; //matches: <name>Kevin</name>  
STRING Fname{xpath('Fname')}; //matches: <Fname>Kevin</Fname>
```

2) Para um campo cujo tipo é uma estrutura RECORD, o XPATH especificado é prefixado em todos os campos nela contidos; caso contrário, o *identifíer* (*identificador*) de caixa baixa do campo seguido de '/' será prefixado nos campos nela contidos. Observe que um XPATH de " (aspas únicas vazias) não prefixarão nada.

```
NameRec := RECORD  
  STRING Fname{xpath('Fname')}; //matches: <Fname>Kevin</Fname>  
  STRING Mname{xpath('Mname')}; //matches: <Mname>Alfonso</Mname>  
  STRING Lname{xpath('Lname')}; //matches: <Lname>Jones</Lname>  
END;  
  
PersonRec := RECORD  
  STRING Uid{xpath('Person[@UID]')};  
  NameRec Name{xpath('Name')};  
  /*matches: <Name>  
    <Fname>Kevin</Fname>  
    <Mname>Alfonso</Mname>  
    <Lname>Jones</Lname>  
  </Name> */  
END;
```

3) Para um campo de DATASET secundário, o XPATH especificado pode ter um dos dois formatos: "Container/Repeated" ou "/Repeated." Cada tag "/Repeated" no Container opcional será iterada para fornecer os valores. Se nenhum XPATH for especificado, o valor padrão do Container será o nome do campo em caixa baixa, e o valor padrão do Repeated será "Row". Por exemplo, isso demonstra "Container/Repeated":

```
DATASET(PeopleNames) People{xpath('people/name')};  
  /*matches: <people>  
    <name>Gavin</name>  
    <name>Ricardo</name>  
  </people> */
```

Isso demonstra "/Repeated":

```
DATASET(Names) Names{xpath('/name')};  
  /*matches: <name>Gavin</name>  
    <name>Ricardo</name> */
```

"Container" e "Repeated" também podem conter filtros xpath, como estes:

```
DATASET(doctorRec) doctors{xpath('person[@job=\'doctor\']')};  
  /*matches: <person job=\'doctor\'>  
    <FName>Kevin</FName>  
    <LName>Richards</LName>  
  </person> */
```

4) Para um campo de tipo SET OF, um xpath em um campo do conjunto pode ter um dos três formatos: "Repeated", "Container/Repeated" ou "Container/Repeated/@attr". Eles são processados de forma semelhante aos datasets, exceto para o seguinte: Se Container for especificado, a leitura XML faz a verificação da tag "Container/All" e, se estiver presente, o conjunto conterá todos os valores possíveis. A terceira forma permite fazer a leitura de valores de atributos em XML.

```
SET OF STRING people;  
  /*matches: <people><All/></people>
```

```
//or: <people><Item>Kevin</Item><Item>Richard</Item></people>

SET OF STRING Npeople{xpath('Name')};
//matches: <Name>Kevin</Name><Name>Richard</Name>
SET OF STRING Xpeople{xpath('/Name/@id')};
//matches: <Name id='Kevin'/><Name id='Richard'/>
```

Para gravar arquivos XML ou JSON usando OUTPUT, as regras são semelhantes com as seguintes exceções:

- Para campos scalar, nomes de tag simples e atributos XML/JSON são suportados.
- Para campos SET, <All> será gerado apenas se o nome do contêiner for especificado.
- Filtros xpath não são suportados.
- A forma "Container/Repeated/@attr" de um SET não é suportada.

Exemplo:

Para DATASET ou o tipo de resultado de uma função TRANSFORM, é preciso especificar apenas o tipo de valor e nome de cada campo no layout:

```
R1 := RECORD
  UNSIGNED1 F1; //only value type and name required
  UNSIGNED4 F2;
  STRING100 F3;
END;

D1 := DATASET('RTTEMP::SomeFile',R1,THOR);
```

Para TABLE de “fatia vertical”, é preciso especificar o tipo de valor, nome e origem de dados para cada campo no layout:

```
R2 := RECORD
  UNSIGNED1 F1 := D1.F1; //value type, name, data source all explicit
  D1.F2; //value type, name, data source all implicit
END;

T1 := TABLE(D1,R2);
```

Para TABLE de relatório de tabela de referência cruzada:

```
R3 := RECORD
  D1.F1; // "group by" fields must come first
  UNSIGNED4 GrpCount := COUNT(GROUP);
  //value type, column name, and aggregate
  GrpSum := SUM(GROUP,D1.F2); //no value type -- defaults to INTEGER
  MAX(GROUP,D1.F2); //no column name in output
END;

T2 := TABLE(D1,R3,F1);
```

```
Form1 := RECORD
  Person.per_last_name; //field name is per_last_name - size
  //is as declared in the person dataset
  STRING25 LocalID := Person.per_first_name;
  //the name of this field is LocalID and it
  //gets its data from Person.per_first_name
  INTEGER8 COUNT(Trades); //this field is unnamed in the output file
  BOOLEAN HasBogey := FALSE;
```

```
                                //HasBogey defaults to false
REAL4      Valu8024;
                                //value from the Valu8024 definition
END;
Form2 := RECORD
    Trades; //include all fields from the Trades dataset at their
            // already-defined names, types and sizes
    UNSIGNED8 fpos {VIRTUAL(fileposition)};
            //contains the relative byte position within the file
END;

Form3 := {Trades, UNSIGNED8 local_fpos {VIRTUAL(localfileposition)}};
        //use of {} instead of RECORD/END
        //"Trades" includes all fields from the dataset at their
        // already-defined names, types and sizes
        //local_fpos is the relative byte position in each part

Form4 := RECORD, MAXLENGTH(10000)
    STRING VarStringName1{MAXLENGTH(5000)};
        //this field is variable size to a 5000 byte maximum

    STRING VarStringName2{MAXLENGTH(4000)};
        //this field is variable size to a 4000 byte maximum

    IFBLOCK(MyCondition = TRUE) //following fields receive values
        //only if MyCondition = TRUE

    BOOLEAN HasLife := TRUE;
        //defaults to true unless MyCondition = FALSE

    INTEGER8 COUNT(Inquiries);
        //this field is zero if MyCondition = FALSE, even
        //if there are inquiries to count

    END;
END;
```

Estruturas de registro embutidas, demonstrando o uso do mesmo nome de campo

```
ds := DATASET('d', { STRING s; }, THOR);
t := TABLE(ds, { STRING60 s := ds.s; });
    // new "s" field is OK with value type explicitly defined
```

Estruturas RECORD de Dataset Filhos

```
ChildRec := RECORD
    UNSIGNED4 person_id;
    STRING20 per_surname;
    STRING20 per_forename;
END;
ParentRecord := RECORD
    UNSIGNED8 id;
    STRING20 address;
    STRING20 CSZ;
    STRING10 postcode;
    UNSIGNED2 numKids;
    DATASET(ChildRec) children{MAXCOUNT(100)};
END;
```

Um exemplo usando {XPATH('tag')}

```
R := record
    STRING10 fname;
    STRING12 lname;
    SET OF STRING1 MySet{XPATH('Set/Element')}; //define set tags
```



```
END;
B := DATASET([{'Fred','Bell',['A','B']},
              {'George','Blanda',['C','D']},
              {'Sam','','['E','F'] } ], R);

OUTPUT(B, '~RTTEST::test.xml', XML);

/* this example produces XML output that looks like this:
<Dataset>
<Row><fname>Fred </fname><lname>Bell</lname>
  <Set><Element>A</Element><Element>B</Element></Set></Row>
<Row><fname>George</fname><lname>Blanda </lname>
  <Set><Element>C</Element><Element>D</Element></Set></Row>
<Row><fname>Sam </fname><lname> </lname>
  <Set><Element>E</Element><Element>F</Element></Set></Row>
</Dataset>
*/
```

Outro exemplo XML com um DATASET filho com 1 campo.

```
cr := RECORD, MAXLENGTH(1024)
  STRING phoneEx{XPATH('')});
END;
r := RECORD, MAXLENGTH(4096)
  STRING id{XPATH('COMP-ID')};
  STRING phone{XPATH('PHONE-NUMBER')};
  DATASET(cr) Fred{XPATH('PHONE-NUMBER-EXP')};
END;

DS := DATASET([{'1002','1352,9493',['1352','9493']},
               {'1003','4846,4582,0779',['4846','4582','0779']}] ,r);

OUTPUT(ds, '~RTTEST::XMLtest2',
  XML('RECORD',
    HEADING('<?xml version="1.0" encoding="UTF-8"?><RECORDS>',
      '</RECORDS>')));

/* this example produces XML output that looks like this:
<?xml version="1.0" encoding="UTF-8"?>
<RECORDS>
  <RECORD>
    <COMP-ID>1002</COMP-ID>
    <PHONE-NUMBER>1352,9493</PHONE-NUMBER>
    <PHONE-NUMBER-EXP>1352</PHONE-NUMBER-EXP>
    <PHONE-NUMBER-EXP>9493</PHONE-NUMBER-EXP>
  </RECORD>
  <RECORD>
    <COMP-ID>1003</COMP-ID>
    <PHONE-NUMBER>4846,4582,0779</PHONE-NUMBER>
    <PHONE-NUMBER-EXP>4846</PHONE-NUMBER-EXP>
    <PHONE-NUMBER-EXP>4582</PHONE-NUMBER-EXP>
    <PHONE-NUMBER-EXP>0779</PHONE-NUMBER-EXP>
  </RECORD>
</RECORDS>
*/
```

XPATH também pode ser usado para definir um arquivo JSON

```
/* a JSON file called "MyBooks.json" contains this data:
[
  {
    "id" : "978-0641723445",
    "name" : "The Lightning Thief",
    "author" : "Rick Riordan"
  }
]
```

```
'
{
  "id" : "978-1423103349",
  "name" : "The Sea of Monsters",
  "author" : "Rick Riordan"
}
]
*/

BookRec := RECORD
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase
  STRING title {XPATH('name')}; //data from name tag, renaming the field
  STRING author; //data from author tag, tag name is lowercase and matches field name
END;

books := DATASET('~jd:mybooks.json',BookRec,JSON('/'));
OUTPUT(books);
```

Ver também: DATASET, DICTIONARY, INDEX, OUTPUT, TABLE, estrutura TRANSFORM, estrutura TYPE, SOAPCALL

DATASET

attr := **DATASET**(*file*, *struct*, *filetype* [, **LOOKUP**]);

attr := **DATASET**(*dataset*, *file*, *filetype* [, **LOOKUP**]);

attr := **DATASET**(**WORKUNIT**([*wuid* ,] *namedoutput*), *struct*);

[*attr* :=] **DATASET**(*recordset* [, *recstruct*]);

DATASET(*row*)

DATASET(*childstruct* [, **COUNT**(*count*) | **LENGTH**(*size*)] [, **CHOSEN**(*maxrecs*)])

[**GROUPED**] [**LINKCOUNTED**] [**STREAMED**] **DATASET**(*struct*)

DATASET(*dict*)

DATASET(*count*, *transform* [, **DISTRIBUTED** | **LOCAL**])

<i>attr</i>	O nome do DATASET a ser usado posteriormente em outras definições.
<i>file</i>	Uma constante da string que contém o nome do arquivo lógico. Consulte a seção <i>Escopo e Nomes de arquivos lógicos</i> para obter mais detalhes sobre nomes de arquivos lógicos.
<i>struct</i>	A estrutura RECORD que define o layout dos campos. Isso deve usar RECORDOF.
<i>filetype</i>	Uma das seguintes palavras-chave, opcionalmente seguidas de opções relevantes àquele tipo de arquivo específico: THOR /FLAT, CSV, XML, JSON, PIPE. Cada uma dessas palavras será abordada em suas próprias seções abaixo.
<i>dataset</i>	Um DATASET previamente definido ou um recordset do qual o layout do registro é derivado. Essa forma é usada principalmente pela ação BUILD e equivale a: <div><pre>ds := DATASET('filename',RECORDOF(anotherdataset), ...)</pre></div>
LOOKUP	Opcional. Especifica que o layout de arquivo deve ser consultado no tempo de compilação. Consulte <i>Resolução de layout de arquivo no tempo de compilação</i> no <i>Guia do Programador</i> para obter mais detalhes.
WORKUNIT	Especifica que o DATASET resulta de um OUTPUT com opção NAMED dentro da mesma ou de outra workunit.
<i>wuid</i>	Opcional. Uma expressão da string que especifica o identificador da workunit ou a workunit que gerou o NAMED OUTPUT.
<i>namedoutput</i>	Uma expressão da string que especifica o nome dado na opção NAMED.
<i>recordset</i>	Um recordset em linha. É possível simplesmente nomear uma definição de conjunto previamente definida, ou usar colchetes explicitamente para indicar uma definição de conjunto em linha. Os registros são separados por vírgulas dentro dos colchetes. Os registros são especificados pelo: 1) Uso de chaves ({ }) circundando os valores do campo de cada registro. Os valores do campo em cada registro são delimitados por vírgula. 2) Uma lista delimitada por vírgula das funções “transform” que geram as linhas de dados. Na lista, todas as funções “transform” devem gerar registros no mesmo formato de resultado.

Referência a Linguagem ECL
Estruturas de registros e arquivos

<i>restruct</i>	Opcional. A estrutura RECORD <i>recordset</i> (<i>conjunto de registros</i>). Pode ser omitida <u>apenas</u> se o parâmetro <i>recordset</i> for de apenas um registro ou uma lista de funções “transform” em linha.
<i>row</i>	Um único registro de dados. Este pode ser um parâmetro especificado em registro único, ou as funções ROW ou PROJECT que definem o dataset de 1 linha.
<i>childstruct</i>	A estrutura RECORD dos registros secundários que estão sendo definidos. Isso pode usar a função RECORDOF.
COUNT	Opcional. Especifica o número de registros secundários anexados ao primário (para ser usado quando servir de interface para os formatos de arquivo externo).
<i>count</i>	Uma expressão que define o número de registros secundários. Pode ser uma constante ou um campo na estrutura RECORD (endereçada como <i>SELF.fieldname</i>).
LENGTH	Opcional. Especifica o <i>tamanho</i> dos registros secundários anexados ao primário (para ser usado quando servir de interface para os formatos de arquivo externo).
<i>size</i>	Uma expressão que define o tamanho dos registros secundários. Pode ser uma constante ou um campo na estrutura RECORD (endereçada como <i>SELF.fieldname</i>).
CHOOSEN	Opcional. Limita o número de registros secundários anexados ao primário. A função CHOOSEN é usada de maneira implícita todas as vezes em que o DATASET filho for lido.
<i>maxrecs</i>	Uma expressão que define o número máximo de registros secundários para um único registro primário.
GROUPED	Especifica que o DATASET que está sendo transferido foi agrupado usando a função GROUP.
LINKCOUNTED	Especifica que o DATASET que está sendo transferido ou retornado usa o formato de contagem de links (cada linha é armazenada como uma alocação de memória individual) em vez do formato padrão (incorporado), onde as linhas do dataset estão todas armazenadas em um único bloco de memória. Isso serve principalmente para usar nas funções BEGNC++ ou funções externas da biblioteca C++.
STREAMED	Especifica que o DATASET que está sendo retornado é retornado como um ponteiro para a interface IRowStream (veja o arquivo de inclusão eclhelper.hpp para a definição). Válido apenas como um tipo de retorno. Isso serve principalmente para usar nas funções BEGNC++ ou funções externas da biblioteca C++.
<i>struct</i>	A estrutura RECORD do campo ou parâmetro do dataset. Isso pode usar a função RECORDOF.
<i>dict</i>	O nome de uma definição do DICTIONARY
<i>count</i>	Uma expressão de número inteiro que especifica a quantidade de registros a ser criada.
<i>transform</i>	A função TRANSFORM que criará os registros. Isso pode levar um parâmetro COUNTER inteiro.
DISTRIBUTED	Opcional. Especifica a distribuição dos registros criados entre todos os nós do cluster. Se omitido, todos os registros são criados no nó 1.
LOCAL	Opcional. Especifica que os registros são criados em cada nó.

A declaração **DATASET** define um arquivo de registros em disco ou na memória. O layout dos registros é especificado por uma estrutura RECORD (os parâmetros *struct* ou *restruct* descritos acima). A distribuição dos registros entre os nós de execução é, de um modo geral, indefinida, pois depende de como será o DATASET (distribuído de uma zona de entrada de arquivos ou gravado em disco por uma ação OUTPUT), do tamanho do cluster em que reside e do tamanho do cluster em que é utilizado (para especificar os requisitos de distribuição para uma operação específica, consulte a função DISTRIBUTE).

As primeiras duas formas são alternativas entre si e ambas podem ser usadas com qualquer um dos *filetypes* descritos abaixo (**THOR/FLAT**, **CSV**, **XML**, **JSON**, **PIPE**).

A terceira forma estabelece o resultado de um OUTPUT com a opção NAMED dentro da mesma tarefa ou da unidade especificada pela *wuid* (consulte **DATASETS** de saída nomeados abaixo).

A quarta forma define um dataset em linha (consulte **DATASETS em linha** abaixo).

A quinta forma é usada somente em um contexto de expressão para permitir o alinhamento de um conjunto de dados de registro único (consulte **Expressões DATASET** de linha única abaixo).

A sexta forma é usada apenas como um tipo de valor em uma estrutura RECORD para definir um DATASET filho (consulte **DATASETS filhos** abaixo).

A sétima forma é usada apenas como um tipo de valor a ser transferido para os parâmetros DATASET (consulte **DATASET como um tipo de parâmetro** abaixo).

A oitava forma é usada para definir um DICTIONARY como um DATASET (consulte **DATASET em DICTIONARY** abaixo).

A nona forma é usada para criar um DATASET usando a função TRANSFORM (consulte **DATASET de TRANSFORM** abaixo)

Arquivos THOR / FLAT

```
attr := DATASET( file, struct, THOR [ __COMPRESSED__ ][,OPT ] [,UNSORTED][,PRELOAD([nbr])][,ENCRYPT(key) ] );
```

```
attr := DATASET( file, struct, FLAT [ __COMPRESSED__ ] [,OPT] [,UNSORTED] [,PRELOAD([nbr])][,ENCRYPT(key) ] );
```

THOR	Especifica que o <i>arquivo</i> está na refinaria de dados (opcionalmente, pode ser especificado como FLAT , que é homólogo do THOR neste contexto).
__COMPRESSED__	Opcional. Especifica que o <i>arquivo</i> THOR está compactado por ser um resultado do serviço do fluxo de trabalho PERSIST por ser um OUTPUT com a opção COMPRESSED.
__GROUPED__	Especifica que o DATASET foi agrupado usando a função GROUP .
OPT	Opcional. Especifica que usar o dataset quando o <i>arquivo</i> THOR não existe, resulta em um conjunto de registro vazio em vez de uma condição de erro.
UNSORTED	Opcional. Especifica que o <i>arquivo</i> THOR não foi classificado, como uma dica para o otimizador.
PRELOAD	Opcional. Especifica que o <i>arquivo</i> é deixado na memória após ter sido carregado (válido apenas quando o Motor de entrega rápida de dados for usado).
<i>nbr</i>	Opcional. Uma constante inteira especificando quantos índices devem ser criados “instantaneamente” para acelerar o acesso ao dataset. Se for > 1.000, especifica a quantidade de memória reservada para esses índices.
ENCRYPT	Opcional. Especifica que o <i>arquivo</i> foi criado pelo OUTPUT com a opção ENCRYPT.
<i>key</i>	Uma constante da string que contém a chave de criptografia usada para criar o arquivo.

Essa forma define um arquivo THOR existente na refinaria de dados. Isso poderia conter registros de comprimento fixo ou variável, dependendo do layout especificado na *struct* do RECORD.

A *struct* pode conter um campo UNSIGNED8 contendo $\{VIRTUAL(fileposition)\}$ ou $\{VIRTUAL(localfileposition)\}$ acrescentado ao nome do campo. Isso indica que o campo possui a posição do registro dentro do arquivo (ou parte), e é usado em instâncias onde é necessário um ponteiro para o registro, como a função BUILD.

Exemplo:

```
PtblRec := RECORD
  STRING2 State := Person.per_st;
  STRING20 City := Person.per_full_city;
  STRING25 Lname := Person.per_last_name;
  STRING15 Fname := Person.per_first_name;
END;

Tbl := TABLE(Person,PtblRec);

PtblOut := OUTPUT(Tbl,,'RTTEMP::TestFile');
//write a THOR file

Ptbl := DATASET('~Thor400::RTTEMP::TestFile',
  {PtblRec,UNSIGNED8 __fpos {VIRTUAL(fileposition)}},
  THOR,OPT);
// __fpos contains the "pointer" to each record
// Thor400 is the scope name and RTTEMP is the
// directory in which TestFile is located
//using ENCRYPT
OUTPUT(Tbl,,'~Thor400::RTTEMP::TestFileEncrypted',ENCRYPT('mykey'));
PtblE := DATASET('~Thor400::RTTEMP::TestFileEncrypted',
  PtblRec,
  THOR,OPT,ENCRYPT('mykey'));
```

Arquivos CSV

attr := DATASET(*file, struct*, CSV [([**HEADING**(*n*)] [, **SEPARATOR**(*f_delimiters*)]

[, **TERMINATOR**(*r_delimiters*)] [, **QUOTE**(*characters*)] [, **ESCAPE**(*esc*)] [, **MAXLENGTH**(*size*)]

[**ASCII** | **EBCDIC** | **UNICODE**] [, **NOTRIM**]) [, **ENCRYPT**(*key*)] [, **__COMPRESSED__**]);

CSV	Especifica que o <i>arquivo</i> corresponde a um arquivo ASCII com valores separados por vírgulas.
HEADING (<i>n</i>)	Opcional. O número de registros de cabeçalho no <i>arquivo</i> . Se omitido, o padrão é zero (0).
SEPARATOR	Opcional. O delimitador do campo. Se omitido, o padrão é uma vírgula (',') ou o delimitador especificado na operação de spray que coloca o arquivo no disco.
<i>f_delimiters</i>	Uma constante de string única, ou conjunto de constantes de string, que define o(s) caractere(s) usado(s) como delimitador de campo. Se as constantes Unicode forem usadas, a representação UTF8 dos caracteres será utilizada.
TERMINATOR	Opcional. O delimitador do registro. Se omitido, o padrão é uma alimentação de linha ('\n') ou o delimitador especificado na operação de spray que coloca o arquivo no disco.
<i>r_delimiters</i>	Uma constante de string única, ou conjunto de constantes de string, que define o(s) caractere(s) usado(s) como delimitador de registro.
QUOTE	Opcional. O caractere de aspas da string utilizado. Se omitido, o padrão é aspas simples (") ou o delimitador especificado na operação de spray que coloca o arquivo no disco.
<i>characters</i>	Uma constante de string única, ou conjunto de constantes de string, que define o(s) caractere(s) usado(s) como delimitador do valor da string.

Referência a Linguagem ECL

Estruturas de registros e arquivos

ESCAPE	Opcional. O caractere de escape da string usado para indicar o caractere seguinte (geralmente um caractere de controle) faz parte dos dados e não deve ser interpretado como um campo ou delimitador de linha. Se omitido, o padrão será o caractere de escape especificado na operação de spray que colocou o arquivo no disco (se for o caso).
<i>esc</i>	Uma constante de string única, ou conjunto de constantes de string, que define os caracteres usados para desvincular os caracteres de controle.
MAXLENGTH (<i>size</i>)	Opcional. Comprimento máximo, em bytes, do registro no <i>arquivo</i> . Se omitido, o padrão é 4096. Há um limite de 10MB mas que pode ser sobrescrito utilizando o comando <i>#OPTION(maxCSVRowSizeMb,nn)</i> onde <i>nn</i> é o tamanho máximo em MB. O tamanho do registro de ser ajustado de maneira mais conservadora possível.
ASCII	Especifica que todas as entradas estão em formato ASCII , incluindo qualquer campo EBCDIC ou UNICODE .
EBCDIC	Especifica que todas as entradas estão em formato EBCDIC, exceto SEPARATOR e TERMINATOR (que são expressados como valores ASCII).
UNICODE	Especifica que todas as entradas estão em formato Unicode UTF8 .
NOTRIM	Especifica que todos os espaços em branco nos dados de entrada estão sendo preservados (o padrão é eliminar os espaços em branco principais).
ENCRYPT	Opcional. Especifica que o <i>arquivo</i> foi criado pelo OUTPUT com a opção ENCRYPT.
<i>key</i>	Uma constante da string que contém a chave de criptografia usada para criar o arquivo.
__COMPRESSED__	Opcional. Especifica que o <i>arquivo</i> está compactado porque correspondia a um OUTPUT com a opção COMPRESSED.

Essa forma é usada para a leitura de um arquivo ASCII CSV. Isso pode ser usado para ler qualquer arquivo de registro de comprimento variável que possua um delimitador de registro definido. Se nenhuma das opções ASCII, EBCDIC, ou UNICODE forem especificadas, a entrada principal estará no formato ASCII com todos os campos UNICODE no formato UTF8.

Exemplo:

```
CSVRecord := RECORD
  UNSIGNED4 person_id;
  STRING20 per_surname;
  STRING20 per_forename;
END;

file1 := DATASET('MyFile.CSV',CSVrecord,CSV);           //all defaults
file2 := DATASET('MyFile.CSV',CSVrecord,CSV(HEADING(1))); //1 header
file3 := DATASET('MyFile.CSV',
  CSVrecord,
  CSV(HEADING(1),
    SEPARATOR([' ','\t']),
    TERMINATOR(['\n','\r\n','\n\r']));
  //1 header record, either comma or tab field delimiters,
  // either LF or CR/LF or LF/CR record delimiters
```

Arquivos XML

attr := **DATASET**(*file, struct, XML*(*xpath* [, **NOROOT**]) [,**ENCRYPT**(*key*)]);

XML	Especifica que o <i>arquivo</i> é um arquivo XML.
<i>xpath</i>	Constante de string que contém um XPATH completo para o tag que delimita os registros no <i>arquivo</i> .

Referência a Linguagem ECL

Estruturas de registros e arquivos

NOROOT	Especifica que o <i>arquivo</i> é um arquivo em XML sem tags de arquivo, apenas com tags de linha.
ENCRYPT	Opcional. Especifica que o <i>arquivo</i> foi criado pelo OUTPUT com a opção ENCRYPT.
<i>key</i>	Uma constante da string que contém a chave de criptografia usada para criar o arquivo.

Essa forma é usada para a leitura de um arquivo XML na refinaria de dados. O parâmetro *xpath* estabelece a tag do delimitador do registro usando um subconjunto da sintaxe padrão do XPATH (www.w3.org/TR/xpath). (Consulte a seção **Suporte ao XPATH** na discussão da estrutura RECORD para obter uma descrição do subconjunto suportado)

O segredo para obter valores de campo individuais do XML está nas definições do campo da estrutura RECORD . Nada de especial será necessário se o nome do campo coincidir exatamente com a tag em caixa baixa do XML que contém os dados. Por outro lado, a adição de *{xpath(xpath tag)}* ao nome do campo (onde *xpath tag* é uma constante de string que possui sintaxe XPATH padrão) requer a extração dos dados. Um XPATH com colchetes angulares vazios (<>) indica que o campo recebe o registro inteiro (completo). Um XPATH absoluto é usado para acessar as propriedades dos elementos principais. Uma vez que XML faz distinção entre maiúsculas e minúsculas, assim como os identificadores ECL , xpaths precisam ser especificados se a tag tiver quaisquer caracteres em caixa alta.

OBSERVAÇÃO: XML a leitura e a análise podem consumir muita memória dependendo do uso. Em particular, se o xpath especificado combinar com uma quantidade muito grande de dados, uma grande estrutura de dados será então fornecida para transformação. Dessa forma, quanto maior for a combinação, mais recursos serão consumidos por combinação. Por exemplo, se você possui um documento muito grande e combinar um elemento próximo da raiz que, virtualmente, engloba o documento todo, o documento inteiro será interpretado como uma estrutura referenciável que o ECL terá acesso.

Exemplo:

```
/* an XML file called "MyFile" contains this XML data:
<library>
  <book isbn="123456789X">
    <author>Bayliss</author>
    <title>A Way Too Far</title>
  </book>
  <book isbn="1234567801">
    <author>Smith</author>
    <title>A Way Too Short</title>
  </book>
</library>
*/

rform := RECORD
  STRING author; //data from author tag -- tag name is lowercase and matches field name
  STRING name {XPATH('title')}; //data from title tag, renaming the field
  STRING isbn {XPATH('@isbn')}; //isbn definition data from book tag
tag
END;

books := DATASET('MyFile',rform,XML('library/book'));
```

Arquivos JSON

attr := DATASET(*file*, *struct*, JSON(*xpath* [, NOROOT]) [ENCRYPT(*key*)]);

Especifica que o	<i>arquivo</i> é um arquivo JSON.
<i>xpath</i>	Constante de string que contém um XPATH completo para o tag que delimita os registros no <i>arquivo</i> .
NOROOT	Especifica que o <i>arquivo</i> é um arquivo JSON sem marcação em nível de raiz, possuindo apenas uma coleção de objetos.

Referência a Linguagem ECL

Estruturas de registros e arquivos

ENCRYPT	Opcional. Especifica que o <i>arquivo</i> foi criado pelo OUTPUT com a opção ENCRYPT.
<i>key</i>	Uma constante da string que contém a chave de criptografia usada para criar o arquivo.

Essa forma é usada para a leitura de um arquivo JSON . O parâmetro *xpath* estabelece o caminho usado para localizar registros no conteúdo JSON usando um subconjunto da sintaxe padrão do standard XPATH (www.w3.org/TR/xpath). (Consulte a seção **Suporte ao XPATH** na discussão da estrutura RECORD para obter uma descrição do subconjunto suportado)

O segredo para obter valores de campo individuais do JSON está nas definições do campo da estrutura RECORD . Nada de especial será necessário se o nome do campo coincidir exatamente com a tag em caixa baixa do XML que contém os dados. Por outro lado, a adição de *{xpath(xpathitag)}* ao nome do campo (onde *xpathitag* é uma constante de string que possui sintaxe XPATH padrão) requer a extração dos dados. Um XPATH com aspas vazias (") indica que o campo recebe o registro inteiro (completo). Um XPATH absoluto é usado para acessar as propriedades dos elementos secundários. Uma vez que XML faz distinção entre maiúsculas e minúsculas, assim como os identificadores ECL, xpaths precisam ser especificados se a tag tiver quaisquer caracteres em caixa alta.

OBSERVAÇÃO: A leitura e a análise do JSON podem consumir muita memória dependendo do uso. Em particular, se o *xpath* especificado combinar com uma quantidade muito grande de dados, uma grande estrutura de dados será então fornecida para transformação. Dessa forma, quanto maior for a combinação, mais recursos serão consumidos por combinação. Por exemplo, se você possui um documento muito grande e combinar um elemento próximo da raiz que, virtualmente, engloba o documento todo, o documento inteiro será interpretado como uma estrutura referenciável que o ECL terá acesso.

Exemplo:

```
/* a JSON file called "MyBooks.json" contains this data:
[
  {
    "id" : "978-0641723445",
    "name" : "The Lightning Thief",
    "author" : "Rick Riordan"
  }
,
  {
    "id" : "978-1423103349",
    "name" : "The Sea of Monsters",
    "author" : "Rick Riordan"
  }
]
*/

BookRec := RECORD
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase
  STRING title {XPATH('name')}; //data from name tag, renaming the field
  STRING author; //data from author tag -- tag name is lowercase and matches field name
END;

books := DATASET('~jd:mybooks.json', BookRec, JSON('/'));
OUTPUT(books);
```

Arquivos PIPE

attr := DATASET(*file*, *struct*, PIPE(*command* [, CSV | XML]));

PIPE	Especifica que o <i>arquivo</i> é originado do comando do <i>programa</i> . Trata-se de um pipe “lido”.
<i>command</i>	O nome do programa a ser executado, que deve gerar registros no formato <i>struct</i> para resultado padrão.

CSV	Opcional. Especifica que o formato dos dados de resultado é CSV. Se omitido, o formato será raw.
XML	Opcional. Especifica que o formato dos dados de resultado é em XML. Se omitido, o formato será raw.

Esta forma usa o PIPE(*comando*) para enviar o *arquivo* ao programa *comando*, que então retorna os registros para o resultado padrão no formato *struct*. Isso também é conhecido como uma entrada PIPE (análoga à função PIPE e opção PIPE em OUTPUT).

Exemplo:

```
PtblRec := RECORD
  STRING2 State;
  STRING20 City;
  STRING25 Lname;
  STRING15 Fname;
END;

Ptbl := DATASET('~Thor50::RTTEMP::TestFile',
  PtblRec,
  PIPE('ProcessFile'));
// ProcessFile is the input pipe
```

DATASETs de saída nomeada

`attr := DATASET(WORKUNIT([wuid ,] namedoutput), struct);`

Esta forma permite usar como um DATASET o resultado de um OUTPUT com a opção NAMED em uma mesma workunit, ou a workunit especificada pela wuid (ID da workunit). *wuid* (workunit ID). Este recurso é mais útil no Motor de entrega rápida de dados.

Exemplo:

```
//Named Output DATASET in the same workunit:
a := OUTPUT(Person(per_st='FL') ,NAMED('FloridaFolk'));
x := DATASET(WORKUNIT('FloridaFolk'),
  RECORDOF(Person));
b := OUTPUT(x(per_first_name[1..4]='RICH'));

SEQUENTIAL(a,b);

//Named Output DATASET in separate workunits:
//First Workunit (wuid=W20051202-155102) contains this code:
MyRec := {STRING1 Value1,STRING1 Value2, INTEGER1 Value3};
SomeFile := DATASET([{'C','G',1},{ 'C','C',2},{ 'A','X',3},
  {'B','G',4},{ 'A','B',5}],MyRec);
OUTPUT(SomeFile,NAMED('Fred'));

// Second workunit contains this code, producing the same result:
ds := DATASET(WORKUNIT('W20051202-155102','Fred'), MyRec);
OUTPUT(ds);
```

DATASET em linhas

`[attr :=] DATASET(recordset , recstruct);`

Esta forma permite alinhar um conjunto de dados para que ele seja tratado como um arquivo. Sua utilidade é justificável em situações onde as operações do arquivo são necessárias em dados gerados de forma dinâmica (tais como valores de tempo de execução de um conjunto de expressões pré-definidas). Também é útil para testar quaisquer condições

de limite das definições ao criar um conjunto de registros bem estabelecidos com que especificamente exercem esses limites. Essa forma pode ser usada em um contexto de expressão.

A estrutura RECORD aninhada, , pode(m) ser representada(s) por registros aninhados dentro de registros. Os DATASETs filhos aninhados também podem ser inicializados dentro das funções TRANSFORM usando datasets em linha (consulte a discussão **DATASETs filhos**).

Exemplo:

```
//Inline DATASET using definition values
myrec := {REAL diff, INTEGER1 reason};
rms5008 := 10.0;
rms5009 := 11.0;
rms5010 := 12.0;
btable := DATASET([ {rms5008,72}, {rms5009,7}, {rms5010,65}], myrec);

//Inline DATASET with nested RECORD structures
nameRecord := {STRING20 lname, STRING10 fname, STRING1 initial := ''};
personRecord := RECORD
    nameRecord primary;
    nameRecord mother;
    nameRecord father;
END;
personDataset := DATASET([ { {'James','Walters','C'},
                              {'Jessie','Blenger'},
                              {'Horatio','Walters'} },
                          { {'Anne','Winston'},
                              {'Sant','Aclause'},
                              {'Elfin','And'} } ], personRecord);

// Inline DATASET containing a Child DATASET
childPersonRecord := {STRING fname, UNSIGNED1 age};
personRecord := RECORD
    STRING20 fname;
    STRING20 lname;
    UNSIGNED2 numChildren;
    DATASET(childPersonRecord) children;
END;

personDataset := DATASET([ { 'Kevin','Hall',2, [ { 'Abby',2 }, { 'Nat',2 } ] },
                          { 'Jon','Simms',3, [ { 'Jen',18 }, { 'Ali',16 }, { 'Andy',13 } ] } ],
                          personRecord);

// Inline DATASET derived from a dynamic SET function
SetIDs(STRING fname) := SET(People(firstname=fname),id);
ds := DATASET(SetIDs('RICHARD'), {People.id});

// Inline DATASET derived from a list of transforms
IDtype := UNSIGNED8;
FMtype := STRING15;
Ltype := STRING25;

resultRec := RECORD
    IDtype id;
    FMtype firstname;
    Ltype lastname;
    FMtype middlename;
END;

T1(IDtype idval, FMtype fname, Ltype lname ) :=
    TRANSFORM(resultRec,
        SELF.id := idval,
```

```
        SELF.firstname := fname,
        SELF.lastname := lname,
        SELF := []);

T2(IDtype idval,FMtype fname,FMtype mname, Ltype lname ) :=
    TRANSFORM(resultRec,
        SELF.id := idval,
        SELF.firstname := fname,
        SELF.middlename := mname,
        SELF.lastname := lname);
ds := DATASET([T1(123,'Fred','Jones'),
               T2(456,'John','Q','Public'),
               T1(789,'Susie','Smith')]);

// You can construct a DATASET from a SET.
SET OF STRING s := ['Jim','Bob','Richard','Tom'];
DATASET(s,{STRING txt});
```

DATASET de linha única

DATASET(row)

Essa forma é usada apenas em um contexto de expressão. Ela permite alinhar um único dataset de registro.

Exemplo:

```
//the following examples demonstrate 4 ways to do the same thing:
personRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
END;

namesRecord := RECORD
    UNSIGNED id;
    personRecord;
END;

namesTable := DATASET('RTTEST::TestRow',namesRecord,THOR);
//simple dataset file declaration form

addressRecord := RECORD
    UNSIGNED id;
    DATASET(personRecord) people; //child dataset form
    STRING40 street;
    STRING40 town;
    STRING2 st;
END;

personRecord tc0(namesRecord L) := TRANSFORM
    SELF := L;
END;

/** 1st way - using in-line dataset form in an expression context
addressRecord t0(namesRecord L) := TRANSFORM
    SELF.people := PROJECT(DATASET([L.id,L.surname,L.forename,L.age]),
                            namesRecord),
                            tc0(LEFT));

    SELF.id := L.id;
    SELF := [];
END;

p0 := PROJECT(namesTable, t0(LEFT));
OUTPUT(p0);
```

```
/** 2nd way - using single-row dataset form
addressRecord t1(namesRecord L) := TRANSFORM
  SELF.people := PROJECT(DATASET(L), tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;

p1 := PROJECT(namesTable, t1(LEFT));
OUTPUT(p1);

/** 3rd way - using single-row dataset form and ROW function
addressRecord t2(namesRecord L) := TRANSFORM
  SELF.people := DATASET(ROW(L, personRecord));
  SELF.id := L.id;
  SELF := [];
END;

p2 := PROJECT(namesTable, t2(LEFT));
OUTPUT(p2);

/** 4th way - using in-line dataset form in an expression context
addressRecord t4(namesRecord l) := TRANSFORM
  SELF.people := PROJECT(DATASET([L], namesRecord), tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;
p3 := PROJECT(namesTable, t4(LEFT));
OUTPUT(p3);
```

DATASETs filhos

DATASET(*childstruct* [**COUNT(*count*) | **LENGTH(*size*)**] [**CHOOSEN(*maxrecs*)**])**

Esta forma é usada como um tipo de valor dentro de uma estrutura RECORD para definir os registros de em um arquivo simples não normalizado. A forma que não contém COUNT ou LENGTH é a mais simples de usar, e simplesmente significa que o dataset, o comprimento e os dados estão armazenados em myfield. A forma COUNT limita o número de elementos à expressão *count*. A forma LENGTH especifica o *tamanho* em outro campo em vez de count. Isso só pode ser usado para entrada de dataset.

As sintaxes alternativas a seguir também são suportadas:

childstruct **fieldname** [SELF.*count*]

DATASET newname := fieldname

DATASET fieldname (forma antiga -- não será mais usada após-SR9)

Qualquer operação pode ser realizada nos DATASETs filhos no hthor e no Motor de entrega rápida de dados (Roxie), porém apenas as operações a seguir são suportadas na refinaria de dados (Thor):

- 1) PROJECT, CHOOSEN, TABLE (não agrupadas), e filtros em tabelas secundárias.
- 2) Operações agregadas são permitidas em qualquer um dos itens acima
- 3) Vários agregados podem ser calculados de uma vez só usando

```
summary := TABLE(x.children, { f1 := COUNT(GROUP),
                                f2 := SUM(GROUP, x),
                                f3 := MAX(GROUP, y) });

summary.f1;
```

- 4) DATASET[n] é compatível com a indexação de elementos secundários

5) SORT(dataset, a, b)[1] também é compatível para encontrar a melhor combinação.

6) Concatenation A de datasets é suportada.

7) TABLEs temporárias também podem ser usadas em conjunção.

8) A inicialização de DATASETs filhos em definições TABLE temporária, permite que [] seja utilizado para inicializar 0 elementos.

Observe que,

```
TABLE(ds, { ds.id, ds.children(age != 10) });
```

não é suportado porque um dataset em uma definição de registro significa “expandir todos os campos do dataset no resultado.” Porém, a adição de um identificador cria uma forma que é suportada:

```
TABLE(ds, { ds.id, newChildren := ds.children(age != 10); });
```

Exemplo:

```
ParentRec := {INTEGER1 NameID, STRING20 Name};
ParentTable := DATASET([ {1, 'Kevin'}, {2, 'Liz'},
                        {3, 'Mr Nobody'}, {4, 'Anywhere'} ], ParentRec);
ChildRec := {INTEGER1 NameID, STRING20 Addr};
ChildTable := DATASET([ {1, '10 Malt Lane'}, {2, '10 Malt Lane'},
                        {2, '3 The cottages'}, {4, 'Here'}, {4, 'There'},
                        {4, 'Near'}, {4, 'Far'} ], ChildRec);

DenormedRec := RECORD
  INTEGER1 NameID;
  STRING20 Name;
  UNSIGNED1 NumRows;
  DATASET(ChildRec) Children;
// ChildRec Children; //alternative syntax
END;

DenormedRec ParentMove(ParentRec L) := TRANSFORM
  SELF.NumRows := 0;
  SELF.Children := [];
  SELF := L;
END;

ParentOnly := PROJECT(ParentTable, ParentMove(LEFT));
DenormedRec ChildMove(DenormedRec L, ChildRec R, INTEGER C) := TRANSFORM
  SELF.NumRows := C;
  SELF.Children := L.Children + R;
  SELF := L;
END;

DeNormedRecs := DENORMALIZE(ParentOnly, ChildTable,
                             LEFT.NameID = RIGHT.NameID,
                             ChildMove(LEFT, RIGHT, COUNTER));
OUTPUT(DeNormedRecs, , 'RTTEMP::TestChildDatasets');

// Using inline DATASET in a TRANSFORM to initialize child records
AkaRec := {STRING20 forename, STRING20 surname};
outputRec := RECORD
  UNSIGNED id;
  DATASET(AkaRec) children;
END;

inputRec := RECORD
  UNSIGNED id;
  STRING20 forename;
  STRING20 surname;
END;
```

```
inPeople := DATASET([
    {1, 'Kevin', 'Halliday'}, {1, 'Kevin', 'Hall'}, {1, 'Gawain', ''},
    {2, 'Liz', 'Halliday'}, {2, 'Elizabeth', 'Halliday'},
    {2, 'Elizabeth', 'MaidenName'}, {3, 'Lorraine', 'Chapman'},
    {4, 'Richard', 'Chapman'}, {4, 'John', 'Doe'}], inputRec);
outputRec makeFatRecord(inputRec l) := TRANSFORM
    SELF.id := l.id;
    SELF.children := DATASET([ l.forename, l.surname ], AkaRec);
END;

fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec l, outputRec r) := TRANSFORM
    SELF.id := l.id;
    SELF.children := l.children + ROW({r.children[1].forename,
                                        r.children[1].surname},
                                        AkaRec);
END;

r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
```

DATASET como um tipo de parâmetro

[GROUPED] [LINKCOUNTED] [STREAMED] DATASET(*struct*)

Esta forma é usada apenas como Tipo de valor para passagem de parâmetros, especificar os tipos de retorno da função ou definir um SET OF (CONJUNTO DE) datasets. Se GROUPED estiver presente, o parâmetro transferido precisa ter sido agrupado através da função GROUP . As palavras-chave LINKCOUNTED e STREAMED são usadas principalmente nas funções BEGINC++ ou nas funções de biblioteca externa C++.

Exemplo:

```
MyRec := {STRING1 Letter};
SomeFile := DATASET([{'A'}, {'B'}, {'C'}, {'D'}, {'E'}], MyRec);

//Passing a DATASET parameter
FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A', 'C', 'E']);
//passed dataset referenced as "ds" in expression

OUTPUT(FilteredDS(SomeFile));

//*****
// The following example demonstrates using DATASET as both a
// parameter type and a return type
rec_Person := RECORD
    STRING20 FirstName;
    STRING20 LastName;
END;

rec_Person_exp := RECORD(rec_Person)
    STRING20 NameOption;
END;

rec_Person_exp xfm_DisplayNames(rec_Person l, INTEGER w) :=
    TRANSFORM
        SELF.NameOption :=
            CHOOSE(w,
                TRIM(l.FirstName) + ' ' + l.LastName,
                TRIM(l.LastName) + ', ' + l.FirstName,
                l.FirstName[1] + l.LastName[1],
                l.LastName);
        SELF := l;
    END;
```

```
DATASET(rec_Person_exp) prototype(DATASET(rec_Person) ds) :=
    DATASET( [], rec_Person_exp );

DATASET(rec_Person_exp) DisplayFullName(DATASET(rec_Person) ds) :=
    PROJECT(ds, xfm_DisplayNames(LEFT,1));

DATASET(rec_Person_exp) DisplayRevName(DATASET(rec_Person) ds) :=
    PROJECT(ds, xfm_DisplayNames(LEFT,2));

DATASET(rec_Person_exp) DisplayFirstName(DATASET(rec_Person) ds) :=
    PROJECT(ds, xfm_DisplayNames(LEFT,3));

DATASET(rec_Person_exp) DisplayLastName(DATASET(rec_Person) ds) :=
    PROJECT(ds, xfm_DisplayNames(LEFT,4));

DATASET(rec_Person_exp) PlayWithName(DATASET(rec_Person) ds_in,
                                     prototype PassedFunc,
                                     STRING1 SortOrder='A',
                                     UNSIGNED1 FieldToSort=1,
                                     UNSIGNED1 PrePostFlag=1) := FUNCTION
FieldPre := CHOOSE(FieldToSort,ds_in.FirstName,ds_in.LastName);
SortedDSPre(DATASET(rec_Person) ds) :=
    IF(SortOrder='A',
        SORT(ds,FieldPre),
        SORT(ds,-FieldPre));
InDS := IF(PrePostFlag=1,SortedDSPre(ds_in),ds_in);

PDS := PassedFunc(InDS); //call the passed function parameter

FieldPost := CHOOSE(FieldToSort,
                    PDS.FirstName,
                    PDS.LastName,
                    PDS.NameOption);
SortedDSPost(DATASET(rec_Person_exp) ds) :=
    IF(SortOrder = 'A',
        SORT(ds,FieldPost),
        SORT(ds,-FieldPost));

OutDS := IF(PrePostFlag=1,PDS,SortedDSPost(PDS));
RETURN OutDS;
END;

//define inline datasets to use.
ds_names1 := DATASET( [{ 'John','Smith'},{'Henry','Jackson'},
                        {'Harry','Potter'}], rec_Person );
ds_names2 := DATASET( [ {'George','Foreman'},
                        {'Sugar Ray','Robinson'},
                        {'Joe','Louis'}], rec_Person );

//get name you want by passing the appropriate function parameter:
s_Name1 := PlayWithName(ds_names1, DisplayFullName, 'A',1,1);
s_Name2 := PlayWithName(ds_names2, DisplayRevName, 'D',3,2);
a_Name := PlayWithName(ds_names1, DisplayFirstName,'A',1,1);
b_Name := PlayWithName(ds_names2, DisplayLastName, 'D',1,1);
OUTPUT(s_Name1);
OUTPUT(s_Name2);
OUTPUT(a_Name);
OUTPUT(b_Name);
```

DATASET de DICIONÁRIO

DATASET(*dict*)

Esta forma redefine o *dict* como um DATASET.

Exemplo:

```
rec := {STRING color, UNSIGNED1 code, STRING name};
ColorCodes := DATASET([{'Black' ,0 , 'Fred'},
                      {'Brown' ,1 , 'Sam'},
                      {'Red'    ,2 , 'Sue'},
                      {'White' ,3 , 'Jo'}], rec);

ColorCodesDCT := DICTIONARY(ColorCodes, {Color, Code});

ds := DATASET(ColorCodesDCT);
OUTPUT(ds);
```

Ver também: OUTPUT, Estrutura RECORD, TABLE, ROW, RECORDOF, Estrutura TRANSFORM, DICTIONARY

DATASET de TRANSFORM

DATASET(*count*, *transform* [, DISTRIBUTED | LOCAL])

Esta forma usa *transform* para criar os registros. O tipo de resultado da função *transform* determina a estrutura. O inteiro COUNTER pode ser usado para numerar cada iteração da função *transform*.

LOCAL é executada separadamente e de forma independente em cada nó.

Exemplo:

```
IMPORT STD;
msg(UNSIGNED c) := 'Rec ' + (STRING)c + ' on node ' + (STRING)(STD.system.Thorlib.Node()+1);

// DISTRIBUTED example
DS := DATASET(CLUSTERSIZE * 2,
              TRANSFORM({STRING line},
                        SELF.line := msg(COUNTER)),
              DISTRIBUTED);
DS;
/* creates a result like this:
  Rec 1 on node 1
  Rec 2 on node 1
  Rec 3 on node 2
  Rec 4 on node 2
  Rec 5 on node 3
  Rec 6 on node 3
*/

// LOCAL example
DS2 := DATASET(2,
               TRANSFORM({STRING line},
                         SELF.line := msg(COUNTER)),
               LOCAL);
DS2;

/* An alternative (and clearer) way
creates a result like this:
  Rec 1 on node 1
  Rec 2 on node 1
  Rec 1 on node 2
  Rec 2 on node 2
  Rec 1 on node 3
```

Rec 2 on node 3
*/

Ver também: Estrutura RECORD, Estrutura TRANSFORM

DICTIONARY

attr := **DICTIONARY**(*dataset*, *structure*);

DICTIONARY(*structure*)

<i>attr</i>	O nome da estrutura DICTIONARY a ser usada posteriormente em outras definições.
<i>dataset</i>	O nome de um DATASET ou conjunto de registros de onde derivar o DICTIONARY . Isso pode ser definido em linha (semelhante a um DATASET em linha).
<i>structure</i>	A estrutura RECORD (muitas vezes, definida em linha) que especifica o layout dos campos. O(s) primeiro(s) campo(s) é(são) campo(s)-chave, seguido(s) opcionalmente pelo operador "resultados em" (=>) e campos adicionais de carga útil. Isso é semelhante à versão da carga útil de um INDEX . A carga útil pode especificar campos individuais ou pode usar o nome do <i>dataset</i> para incluir todos os campos não chave no payload.

Um **DICTIONARY** permite verificar eficientemente se um determinado valor de dados está em uma lista (usando o operador **IN**) ou simplesmente mapear dados. Isso é semelhante a um **LOOKUP JOIN** que pode ser usado em qualquer contexto.

Definição **DICTIONARY**

A declaração **DICTIONARY** define um conjunto de registros únicos derivados do parâmetro *dataset* e indexado pelo(s) primeiro(s) campo(s) nomeado(s) no parâmetro *structure* . O **DICTIONARY** contém um registro para cada valor único no(s) campo(s)-chave. Você pode acessar um registro individual anexando colchetes ([]) ao nome do *attr* do **DICTIONARY**, que contém o(s) valor(es) do(s) campo(s)-chave que identifica(m) o registro específico a ser acessado.

DICTIONARY como Tipo de Valor

A segunda forma do **DICTIONARY** é um tipo de valor com o parâmetro *structure* especificando a estrutura **RECORD** dos dados. Esse tipo de dados permite especificar um **DICTIONARY** como um dataset filho, de forma semelhante a como **DATASET** pode ser usado para definir um dataset filho. Isso também pode ser usado para passar um **DICTIONARY** como parâmetro.

Exemplo:

```
ColorCodes := DATASET([ { 'Black' , 0 },
                        { 'Brown' , 1 },
                        { 'Red'    , 2 },
                        { 'Orange' , 3 },
                        { 'Yellow' , 4 },
                        { 'Green'  , 5 },
                        { 'Blue'   , 6 },
                        { 'Violet' , 7 },
                        { 'Grey'   , 8 },
                        { 'White'  , 9 } ], { STRING color, UNSIGNED1 code });

ColorCodesDCT := DICTIONARY(ColorCodes, { Color, Code }); //multi-field key
ColorCodeDCT  := DICTIONARY(ColorCodes, { Color => Code }); //payload field
CodeColorDCT  := DICTIONARY(ColorCodes, { Code => Color });

//mapping examples
MapCode2Color(UNSIGNED1 code) := CodeColorDCT[code].color;
MapColor2Code(STRING color)  := ColorCodeDCT[color].code;
```

Referência a Linguagem ECL

Estruturas de registros e arquivos

```
OUTPUT(MapColor2Code('Red'));           //2
OUTPUT(MapCode2Color(4));               //'Yellow'

//Search term examples
OUTPUT('Green' IN ColorCodeDCT);        //true
OUTPUT(6 IN CodeColorDCT);              //true
OUTPUT(ROW({'Red',2},RECORDOF(ColorCodes)) IN ColorCodesDCT); //multi-field key, true

//multi-field payload examples
rec := RECORD
  STRING10 color;
  UNSIGNED1 code;
  STRING10 name;
END;
Ds := DATASET([{'Black' ,0 , 'Fred'},
               {'Brown' ,1 , 'Seth'},
               {'Red'   ,2 , 'Sue'},
               {'White' ,3 , 'Jo'}], rec);

DsDCT := DICTIONARY(DS,{color => DS});

OUTPUT('Red' IN DsDCT); //true
DsDCT['Red'].code;      //2
DsDCT['Red'].name;      //Sue

//inline DCT examples
InlineDCT := DICTIONARY([{'Black' => 0 , 'Fred'},
                        {'Brown' => 1 , 'Sam'},
                        {'Red'   => 2 , 'Sue'},
                        {'White' => 3 , 'Jo'} ],
                        {STRING10 color => UNSIGNED1 code,STRING10 name});

OUTPUT('Red' IN InlineDCT); //true
InlineDCT['Red'].code;      //2
InlineDCT['Red'].name;      //Sue
InlineDCT['Red'];           //Red 2 Sue

//Form 2 examples -- parameter passing
MyDCTfunc(DICTIONARY({STRING10 color => UNSIGNED1 code,STRING10 name}) DCT,
          STRING10 key) := DCT[key].name;
MyDCTfunc(InlineDCT,'White'); //Jo
MyDCTfunc(DsDCT,'Brown');     //Seth
```

Ver também: DATASET, Estrutura RECORD, INDEX, Operador IN

INDEX

attr := INDEX([*baserecset*,] *keys*, *indexfile* [,SORTED] [,OPT] [,COMPRESSED(LZW | ROW | FIRST)] [,DISTRIBUTED] [,FILEPOSITION([*flag*])] [, MAXLENGTH(***value***)]);

attr := INDEX([*baserecset*,] *keys*, *payload*, *indexfile* [,SORTED] [,OPT] [,COMPRESSED(LZW | ROW | FIRST)] [,DISTRIBUTED] [,FILEPOSITION([*flag*])] [, MAXLENGTH(*value*)]);

attr := INDEX(*index*,*newindexfile* [, MAXLENGTH(*value*)]);

<i>attr</i>	O nome do INDEX a ser usado posteriormente em outros atributos.
<i>baserecset</i>	Opcional. O conjunto de registros de dados para o qual o arquivo de índice foi criado. Se omitido, todos os campos dos parâmetros <i>keys</i> e <i>payload</i> devem ser totalmente qualificados. >
<i>keys</i>	A estrutura RECORD dos campos no <i>indexfile</i> que contém informações-chave e de posição do arquivo a serem mencionadas no <i>baserecset</i> . Os nomes e tipos de campos devem corresponder aos campos de <i>baserecset</i> (os tipos de campos REAL e DECIMAL não são permitidos). Isso também pode conter campos adicionais que não estão presentes no <i>baserecset</i> (campos calculados). Se omitido, todos os campos em <i>baserecset</i> serão usados.
<i>payload</i>	A estrutura RECORD do <i>indexfile</i> que contém campos adicionais não usados como chaves. O nome do atributo. <i>baserecset</i> se o nome do baserecset estiver na estrutura, especificará "todos os outros campos ainda não nomeados no parâmetro <i>keys</i> ". Isso pode conter campos que não estão presentes no <i>baserecset</i> (campos calculados). A <i>payload</i> esses campos não ocupam espaço nos nós non-leaf do índice e não podem ser citados em uma cláusula de filtro KEYED() Todos os campos com o modificador {BLOB} (para permitir mais de 32 K de dados por entrada de índice) são armazenados no <i>indexfile</i> , mas não com o resto do registro. O acesso aos dados do BLOB exige uma procura adicional.
<i>indexfile</i>	Uma constante de string que contém o nome do arquivo lógico do índice. Consulte a seção <i>Escopo e Nomes de arquivos lógicos</i> para obter mais detalhes sobre nomes de arquivos lógicos.
SORTED	Opcional. Especifica que, quando o índice é acessado, os registros são fornecidos na ordem das <i>keys</i> . Se omitido, a ordem do registro retornado será indefinida.
OPT	Opcional. Significa que usar o índice quando o <i>indexfile</i> não existe resulta em um conjunto de registros vazio em vez de em uma condição de erro.
COMPRESSED	Opcional. Especifica o tipo de compactação usado. Se omitido, o padrão usado é LZW , uma variante do algoritmo Lempel-Ziv-Welch. A especificação de ROW compacta entradas de índice com base nas diferenças entre linhas contíguas (apenas para uso com registros de comprimento fixo) e é recomendada quando um tempo de descompactação rápido é mais importante que a quantidade de compactação conseguida. FIRST compacta os elementos principais comuns da chave (recomendado apenas para uso na comparação de cronometragem de tempo).
DISTRIBUTED	Opcional. Especifica que o índice foi criado com a opção DISTRIBUTED na ação BUILD ação ou que a ação BUILD simplesmente fez referência à declaração INDEX com a opção DISTRIBUTED. Portanto, o INDEX é acessado localmente em cada nó (de forma semelhante à função LOCAL, que é preferível), não é classificado globalmente e não há um índice raiz para indicar que parte do índice conterá uma determinada entrada. Isso pode ser útil em queries do Roxie juntamente com o uso de ALLNODES.

Referência a Linguagem ECL
Estruturas de registros e arquivos

FILEPOSITION	Opcional. Se a <i>flag</i> for FALSE, evitará o comportamento padrão da criação do campo implícito fileposition e não tratará um campo inteiro à direita de forma diferente do resto da carga útil.
<i>flag</i>	Opcional. TRUE ou FALSE, indicando se o campo “fileposition” implícito será ou não criado.
<i>index</i>	O nome de um atributo INDEX previamente definido a ser duplicado.
<i>newindexfile</i>	Uma constante de string contendo o nome do arquivo lógico do novo índice. Consulte a seção <i>Escopo e Nomes de arquivos lógicos</i> para obter mais detalhes sobre nomes de arquivos lógicos.
MAXLENGTH	Opcional. Esta opção é usada para criar índices que são compatíveis com versões anteriores às versões 3.0. Especifica o comprimento máximo de um registro de índice de comprimento variável. Os registros de comprimento fixo sempre utilizam o tamanho mínimo exigido. Se o comprimento máximo padrão causar problemas de ineficiência, ele pode ser substituído de forma explícita.
<i>value</i>	Opcional. Um valor inteiro que indica o comprimento máximo. Se omitido, o tamanho máximo será calculado a partir da estrutura do registro. Registros de comprimento variável que não especificam MAXLENGTH podem ser ligeiramente ineficientes.

INDEX declara um índice previamente criado para uso. INDEX é relacionado a BUILD (ou BUILDINDEX) da mesma forma que DATASET é relacionado a OUTPUT: BUILD cria um arquivo de índice que INDEX define para uso no código ECL code. Os arquivos de índice são compactados. Um único registro de índice deve ser definido como menos de 32K e resultar em uma página de menos de 8 K após a compactação.

Na parte de metachave da árvore Binária de INDEX, há uma parte do arquivo separada de 32K no primeiro nó do cluster Thor onde foi criado, mas que foi implementada em todos os nós de um cluster Roxie. O número de partes de arquivo do nó folha é igual ao número de nós do cluster Thor onde foi criado. No geral, a distribuição específica de registros de nó folha entre nós de execução é indefinida, pois depende do tamanho do cluster onde foi criado e do tamanho do cluster onde é usado.

Esses tipos de dados são permitidos na parte das chaves de um INDEX:

- BOOLEAN
- INTEGER
- UNSIGNED
- STRING
- DATA
- QSTRING

Todas as STRINGS devem ter comprimento fixo.

INDEX de acesso chaveado

Essa forma define um arquivo de índice para permitir acesso ao *baserecset* usando chaves. O índice é usado principalmente pelas operações FETCH e JOIN (com a opção KEYED option).

Exemplo:

```
PtblRec := RECORD
  STRING2 State := Person.per_st;
```

```
STRING20 City := Person.per_full_city;
STRING25 Lname := Person.per_last_name;
STRING15 Fname := Person.per_first_name;
END;

PtblOut := OUTPUT(TABLE(Person,PtblRec),, 'RTTEMP::TestFetch');

Ptbl := DATASET('RTTEMP::TestFetch',
  {PtblRec,UNSIGNED8 RecPtr {VIRTUAL(fileposition)}} ,
  FLAT);

AlphaInStateCity := INDEX(Ptbl,
  {state,city,lname,fname,RecPtr},
  'RTTEMPkey::TestFetch');

Bld := BUILDINDEX(AlphaInStateCity);
```

INDEX Payload

Essa forma define um arquivo de índice que contém campos adicionais de carga útil, além das chaves. A carga útil pode conter campos com o modificador {BLOB} para permitir a criação de mais de 32 K de dados por entrada de índice. Esses campos BLOB são armazenados dentro do *indexfile*, mas não com o resto do registro. O acesso aos dados do BLOB exige uma procura adicional.

Essa forma é usada principalmente em operações JOIN com “half-key” para eliminar a necessidade de acessar diretamente o *baserecset*. Assim, o desempenho é superior ao da versão da mesma operação com “chave completa” (executada com a opção KEYED no JOIN). Por padrão, os campos de carga útil não são classificados durante a ação BUILD para minimizar o espaço dos nós folha da chave. Esse comportamento de classificação pode ser controlado usando *sortIndexPayload* em uma declaração #OPTION.

Exemplo:

```
Vehicles := DATASET('vehicles',
  {STRING2 st,STRING20 city,STRING20 lname,
  UNSIGNED8 fpos{VIRTUAL(fileposition)}} ,FLAT);

VehicleKey := INDEX(Vehicles,{st,city},{lname,fpos}, 'vkey::st.city');
BUILDINDEX(VehicleKey);
```

INDEX Duplicado

Essa forma define um *newindexfile* idêntico ao *index* previamente definido.

Exemplo:

```
NewVehicleKey := INDEX(VehicleKey, 'NEW::vkey::st.city');
//define NewVehicleKey like VehicleKey
```

Ver também: DATASET, BUILDINDEX, JOIN, FETCH, KEYED/WILD

Escopo e Nomes de Arquivo Lógicos.

Escopo do Arquivo

Os nomes de arquivo lógicos usados nos atributos DATASET e INDEX e as ações OUTPUT e BUILD (ou BUILDINDEX) podem começar opcionalmente com um ~, o que significa que são absolutos. Caso contrário, são relativos (o prefixo de escopo configurado para a plataforma é acrescentado ao início do nome do arquivo). Ele pode conter escopos delimitados por dois caracteres de dois pontos (::), e a parte final é o nome do arquivo. Não é permitido ter dois caracteres de dois pontos (::) no final do nome do arquivo. É possível especificar um qualificador de cluster. Por exemplo, ~myfile@mythor2 aponta para um arquivo em que o arquivo está em vários clusters no mesmo escopo. Os caracteres ASCII válidos de um escopo ou nome de arquivo devem ser maiores que 32 e menores que 127, exceto * / : < > ? e |.

Para fazer referência a caracteres maiúsculos em caminhos e nomes de arquivos físicos, use o caractere circunflexo (^). Por exemplo, ~file::10.150.254.6::var::lib::^h^p^c^systems::mydropzone::^people.txt".

A presença de um escopo no nome do arquivo permite substituir o nome do escopo padrão do cluster. Por exemplo, supondo que você está operando em um cluster com nome de escopo padrão "Training", as duas ações OUTPUT a seguir resultarão no mesmo escopo:

```
OUTPUT(SomeFile,, 'SomeDir::SomeFileOut1');  
OUTPUT(SomeFile,, '~Training::SomeDir::SomeFileOut2');
```

A presença de um til no início do nome do campo define apenas o nome do escopo e não altera o conjunto de discos onde os dados são gravados (os arquivos são **sempre gravados nos discos do cluster onde o código é executado**). A declaração DATASETs para esses arquivos pode ser parecida com:

```
RecStruct := {STRING line};  
ds1 := DATASET('SomeDir::SomeFileOut1',RecStruct,THOR);  
ds2 := DATASET('~Training::SomeDir::SomeFileOut2',RecStruct,THOR);
```

Esses dois arquivos estão no mesmo escopo. Portanto, quando você usar DATASETs em uma tarefa, o utilitário de arquivos distribuídos (DFU) procurará os dois arquivos no escopo Training.

No entanto, uma vez que você souber o nome do escopo, poderá fazer referência a arquivos de qualquer outro cluster dentro do mesmo ambiente. Por exemplo, supondo que você está operando em um cluster com o nome de escopo padrão "Production" e quer usar os dados nos dois arquivos acima. As duas definições de DATASET a seguir permitem acessar esses dados:

```
FileX := DATASET('~Training::SomeDir::SomeFileOut1',RecStruct,THOR);  
FileY := DATASET('~Training::SomeDir::SomeFileOut2',RecStruct,THOR);
```

Note a presença do nome do escopo nas duas definições. Isso é obrigatório porque os arquivos estão em outro escopo.

Arquivos Estrangeiros

De forma semelhante às regras de escopo descritas acima, você também pode referenciar arquivos em ambientes separados atendidos por um Dali diferente. Isso permite uma referência somente leitura a arquivos (super e lógicos) remotos.

OBSERVAÇÃO: Se a autenticação LDAP estiver ativada no Dali remoto, as credenciais do usuário serão verificadas antes do processamento da solicitação de acesso ao arquivo. Se a segurança de escopo de arquivo do LDAP estiver habilitada no Dali remoto, as permissões de acesso aos arquivos do usuário também serão verificadas.

A sintaxe é semelhante a:

'~foreign::<dali-ip>::<scope>::<tail>'

Por exemplo,

```
MyFile :=DATASET('~foreign::10.150.50.11::training::thor::myfile',  
RecStruct,FLAT);
```

concede acesso somente leitura ao arquivo remoto *training::thor::myfile* no ambiente *10.150.50.11*.

Arquivos da Zona de Entrada

Você também pode ler e gravar arquivos diretamente em uma zona de entrada de arquivos (ou qualquer outra caixa endereçável por IP) que não foi sprayed para o Thor. A zona de entrada de arquivos deve estar executando o programa utilitário *dafileserv*. Se o host for um host Windows, *dafileserv* deve ser instalado como serviço.

A sintaxe é semelhante a:

'~file::<LZ-ip>::<path>::<filename>'

Por exemplo,

```
MyFile :=DATASET('~file::10.150.50.12::c$::training::import::myfile',RecStruct,FLAT);
```

concede acesso ao arquivo remoto *c\$/AdvancedECL/myfile* na zona de entrada de arquivos *10.150.50.12* baseada em Linux.

Os nomes de arquivo lógicos da ECL não fazem distinção entre maiúsculas e minúsculas. Os nomes físicos são, por padrão, em minúsculas, o que pode causar problemas quando a zona de entrada de arquivos é um host Linux (o Linux faz distinção entre maiúsculas e minúsculas). Os caracteres podem ser convertidos explicitamente em maiúsculas acrescentando o caractere de escape circunflexo (^) ao início do nome, como neste exemplo:

```
MyFile :=DATASET('~file::10.150.50.12::c$::^Advanced^E^C^L::myfile',RecStruct,FLAT);
```

concede acesso ao arquivo remoto *c\$/AdvancedECL/myfile* na zona de entrada de arquivos *10.150.50.12* baseada em Linux.

Arquivos Dinâmicos

Em queries do Roxie (e somente nelas), você também pode ler arquivos que podem não existir no momento da implementação, mas que existirão no tempo de execução da query, tornando o nome de arquivo DYNAMIC.

A sintaxe é semelhante a:

DYNAMIC('<filename>')

Por exemplo,

```
MyFile :=DATASET(DYNAMIC('~training::import::myfile'),RecStruct,FLAT);
```

Isso faz com que o arquivo seja resolvido na execução da query e não na implementação.

Arquivos Temporários

Um Superarquivo é uma coleção de arquivos lógicos tratada como única entidade (consulte o artigo **Visão geral dos Superarquivos** no *Guia do Programador*). Você pode especificar um Superarquivo temporário nomeando o conjunto de subarquivos entre chaves na string que nomeia o arquivo lógico da declaração DATASET. A sintaxe é semelhante a:

```
DATASET('{ listoffiles }', recstruct, THOR);
```

listoffiles Uma lista delimitada por vírgulas do conjunto de arquivos lógicos a serem tratados como um único Super Arquivo. Os nomes de arquivos lógicos devem seguir as regras listadas acima, com uma exceção: o til que indica substituição de nome de escopo pode ser especificado em cada nome na lista ou fora das chaves.

Por exemplo, supondo que o nome de escopo padrão é "thor", os exemplos a seguir definem o mesmo Super Arquivo:

```
MyFile :=DATASET('{in::file1,
                  in::file2,
                  ~train::in::file3}'),
          RecStruct,THOR);

MyFile :=DATASET('~{thor::in::file1,
                  thor::in::file2,
                  train::in::file3}'),
          RecStruct,THOR);
```

Não é possível usar essa forma de nome de arquivo lógico para executar um OUTPUT ou PERSIST, pois ela é somente leitura.

Racionalidade Implícitos de Dataset

Dataset filhos aninhados em uma Refinaria de dados (Thor) ou em um cluster de Motor de entrega rápida de dados (Roxie) são inerentemente relacionais, pois todos os dados principais-secundários estão contidos em um único registro físico. As seguintes regras se aplicam a todas as relações inerentes.

O nível de escopo de uma determinada query é definido pelo dataset pai da query. Durante a query, supõe-se que você está trabalhando com um único registro desse dataset pai.

Vamos supor que você tem a seguinte estrutura relacional no banco de dados:

Household	Parent
Person	Child of Household
Accounts	Child of Person, Grandchild of Household

Isso significa que, no nível do escopo pai:

a) Todos os campos de qualquer arquivo que têm uma relação de um para muitos com o arquivo pai estarão disponíveis. Ou seja, todos os campos em qualquer registro principal (ou registro principal de um registro principal etc.) estão disponíveis para o secundário. Por exemplo, se o dataset Person for o escopo pai, todos os campos do dataset Household estarão disponíveis.

b) Todos os datasets secundários (ou datasets filhos de um Dataset filho etc.) podem ser usados para filtrar o principal, desde que a subquery use uma função agregada ou opere no nível de existência de um datasets filhos que cumpram os critérios do filtro (consulte EXISTS). Você pode usar campos específicos de dentro de um registro filho no nível de escopo do registro principal usando EVALUATE ou subcrevendo ([]) um registro filho específico. Por exemplo, se o dataset Person for o escopo pai, você poderá filtrar o conjunto de registros relacionados de Accounts e verificar se filtrou todos os registros relacionados de Accounts.

c) Se um dataset for usado em um escopo em que não é um filho do dataset pai, será avaliado no escopo que abrange o dataset. Por exemplo, a expressão:

```
Household(Person(personage > AVE(Person, personage)))
```

significa "residências que contêm pessoas com idade acima da idade média da residência". Isso não significa "residências que contêm pessoas com idade acima da idade média de todas as residências". Isso ocorre porque o dataset principal (Household) abrange o dataset filho (Person), o que faz com que a avaliação da função AVE opere no nível das pessoas dentro da residência.

d) Um atributo definido com o serviço de fluxo de trabalho STORED() é avaliado no nível global. É um erro se não puder ser avaliado independentemente de outros datasets. Isso pode levar a alguns comportamentos um pouco estranhos:

```
AveAge := AVE(Person, personage);  
MyHouses := Household(Person(personage > aveAge));
```

significa "residências que contêm pessoas com idade acima da idade média da residência". No entanto,

```
AveAge := AVE(Person, personage) : STORED('AveAge');  
MyHouses := Household(Person(personage > aveAge));
```

Significa "residências que contêm pessoas com idade acima da idade média de todas as residências". Isso ocorre porque agora o atributo AveAge é avaliado fora do escopo que abrange Household.

Tipos de Dados Não Nativos

Estrutura TYPE

TypeName := TYPE

functions;

END;

<i>TypeName</i>	O nome da Estrutura TYPE.
<i>functions</i>	Definições do atributo da função. Geralmente existem múltiplas <i>funções</i> .

A estrutura **TYPE** define uma série de *funções* que são invocadas de forma implícita quando o *TypeName* é subsequentemente usado em uma estrutura **RECORD** como tipo de valor. Os parâmetros podem ter sido especificados para o atributo da estrutura **TYPE**, que podem então ser usados em qualquer definição da *função*. Para especificar os parâmetros, basta anexá-los ao *TypeName* usado na estrutura **RECORD** para definir o tipo de valor para o campo.

Uma estrutura **TYPE** pode conter apenas definições de função contidas na lista de Funções especiais disponíveis (consulte **Funções especiais da estrutura TYPE**).

Exemplo:

```
STRING4 Rev(STRING4 S) := S[4] + S[3] + S[2] + S[1];
EXPORT ReverseString4 := TYPE
    EXPORT STRING4 LOAD(STRING4 S) := Rev(S);
    EXPORT STRING4 STORE(STRING4 S) := Rev(S);
END;
NeedC(INTEGER len) := TYPE
    EXPORT STRING LOAD(STRING S) := 'C' + S[1..len];
    EXPORT STRING STORE(STRING S) := S[2..len+1];
    EXPORT INTEGER PHYSICALLength(STRING S) := len;
END;
ScaleInt := TYPE
    EXPORT REAL LOAD(INTEGER4 I) := I / 100;
    EXPORT INTEGER4 STORE(REAL R) := ROUND(R * 100);
END;
R := RECORD
    ReverseString4 F1;
    // Defines a field size of 4 bytes. When R.F1 is used,
    // the ReverseString4.Load function is called passing
    // in those four bytes and returning a string result.
    NeedC(5) F2;

    // Defines a field size of 5 bytes. When R.F2 is used,
    // those 5 bytes are passed in to NeedC.Load (along with
    // the length 5) and a 6 byte string is returned.
    ScaleInt F3;

    // Defines a field size of 4. When R.F3 is used, the
    // ScaleInt.Load function returns the number / 100.
END;
```

Ver também: Estrutura **RECORD**, Funções especiais da estrutura **TYPE**

Funções Especiais da Estrutura TYPE

LOAD

EXPORT *LogicalType* **LOAD**(*PhysicalType alias*) := *expression*;

<i>LogicalType</i>	O tipo de valor do resultado provido da função.
<i>PhysicalType</i>	O tipo de valor do parâmetro de entrada para a função.
<i>alias</i>	O nome da entrada a ser usada na <i>expressão</i> .
<i>expression</i>	A operação a ser desempenhada na entrada.

LOAD define a função de retorno do acionamento a ser aplicada aos bytes do registro para criar o valor de dados a ser usado no cálculo. Esta função define como o sistema faz a leitura dos dados do disco.

STORE

EXPORT *PhysicalType* **STORE**(*LogicalType alias*) := *expression*;

<i>PhysicalType</i>	O tipo de valor do resultado provido da função.
<i>LogicalType</i>	O tipo de valor do parâmetro de entrada para a função.
<i>alias</i>	O nome da entrada a ser usada na <i>expressão</i> .
<i>expression</i>	A operação a ser desempenhada na entrada.

STORE define a função de retorno do acionamento a ser aplicada ao valor calculado para armazená-la no registro. Esta função define como o sistema grava os dados no disco.

PHYSICALENGTH

EXPORT INTEGER **PHYSICALENGTH**(*type alias*) := *expression*;

<i>type</i>	O tipo de valor do parâmetro de entrada para a função.
<i>alias</i>	O nome da entrada a ser usada na <i>expressão</i> .
<i>expression</i>	A operação a ser desempenhada na entrada.

PHYSICALENGTH define a função de retorno do acionamento para determinar os requisitos de armazenagem do formato lógico no formato físico especificado. Esta função define quantos bytes os dados ocupam no disco.

MAXLENGTH

EXPORT INTEGER **MAXLENGTH** := *expression*;

<i>expression</i>	Uma constante inteira que define o comprimento físico máximo dos dados.
-------------------	---

MAXLENGTH define a função de retorno do acionamento para determinar o comprimento físico máximo dos dados de comprimento variável.

GETISVALID

EXPORT BOOLEAN **GETISVALID**(*PhysicalType alias*) := *expression*;

Referência a Linguagem ECL

Tipos de Dados Não Nativos

<i>PhysicalType</i>	O tipo de valor do parâmetro de entrada para a função.
<i>alias</i>	O nome da entrada a ser usada na <i>expressão</i> .
<i>expression</i>	A operação a ser desempenhada na entrada.

GETISVALID define a função de retorno do acionamento para determinar quais valores de dados estão no formato físico especificado.

Exemplo:

```
EXPORT NeedC(INTEGER len) := TYPE
  EXPORT STRING LOAD(STRING S) := 'C' + S[1..len];
  EXPORT STRING STORE(STRING S) := S[2..len+1];
  EXPORT INTEGER PHYSICALENGTH(STRING S) := len;
  EXPORT INTEGER MAXLENGTH(STRING S) := len;
  EXPORT BOOLEAN GETISVALID(STRING S) := S[1] <> 'C';
END;

// delimited string data type
EXPORT dstring(STRING del) := TYPE
  EXPORT INTEGER PHYSICALENGTH(STRING s) :=
    Std.Str.Find(s,del)+length(del)-1;
  EXPORT STRING LOAD(STRING s) :=
    s[1..Std.Str.Find(s,del)-1];
  EXPORT STRING STORE(STRING s) := s + del;
END;
```

See Also: Estrutura TYPE

Suporte a Análise (Parsing)

Suporte a Análise (Parsing)

Análise de Linguagem Natural é realizada no ECL através da combinação de definições do padrão com uma estrutura RECORD gerada (ou função TRANSFORM) projetada especificamente para receber os valores analisados, e depois usar a função PARSE para realizar a operação.

As definições do padrão são usadas para detectar o texto "interessante" dentro dos dados. Assim como ocorre com todas as outras definições de atributo, esses padrões normalmente definem elementos de avaliação específicos e podem ser combinados para formar padrões, tokens e regras mais complexas.

A estrutura RECORD gerada (ou função TRANSFORM) define o formato do recordset resultante. Ela normalmente possui funções específicas de correspondência de padrão que retornam o texto “interessante”, seu comprimento ou posição.

A função PARSE implementa a operação de análise. Ela retorna um recordset que pode ser processado posteriormente, conforme necessário, usando a sintaxe padrão do ECL ou simplesmente o resultado.

Tipo de Valores PARSE Pattern

Há três tipos de valores especificamente projetados e necessários para definir os atributos do padrão de análise:

PATTERN *patternid* := *parsepattern*;

<i>patternid</i>	O nome do atributo do padrão.
<i>parsepattern</i>	O padrão, bastante semelhante às expressões regulares. Pode conter outros atributos PATTERN previamente definidos. Consulte Definições ParsePattern abaixo .

O tipo de valor **PATTERN** define uma expressão de análise bastante semelhante aos padrões de uma .

TOKEN *tokenid* := *parsepattern*;

<i>tokenid</i>	O nome do atributo do token.
<i>parsepattern</i>	O padrão do token, bastante semelhante às expressões regulares. Isso pode conter atributos PATTERN, mas nenhum atributo TOKEN ou RULE. Consulte Definições ParsePattern abaixo .

O tipo de valor **TOKEN** define uma expressão de análise bastante semelhante a um PATTERN, mas uma vez combinado, o analisador não recua para encontrar correspondências alternativas como faria com PATTERN.

RULE [(*reconstruct*)] *ruleid* := *rulePattern*;

<i>reconstruct</i>	Opcional. O nome do atributo de uma estrutura RECORD (válido apenas quando a opção PARSE é usada na função PARSE).
<i>ruleid</i>	O nome do atributo da regra.
<i>rulePattern</i>	O padrão da regra, bastante semelhante às expressões regulares. Isso pode conter atributos PATTERN, mas nenhum atributo TOKEN ou RULE. Consulte Definições ParsePattern abaixo .

O tipo de valor **RULE** define uma expressão de análise que contém combinações de TOKENs. Se uma definição RULE contiver um PATTERN será implicitamente convertido em um TOKEN. Assim como PATTERN, uma vez combinado, o analisador recua para buscar correspondências alternativas RULE.

Se a opção PARSE estiver presente na função PARSE (implementando análise tomata para a operação), cada rulePattern alternativa RULE *rulePattern* deve apresentar uma função TRANSFORM associada. Os diferentes padrões de entrada podem ser referidos usando \$1, \$2, etc. Se o padrão possui um *reconstruct* associado, \$1 será uma linha; caso contrário, será uma string. As funções TRANSFORM padrão são criadas em duas circunstâncias:

1. Se não houver padrão, a transform padrão limpará a linha. Por exemplo:

```
RULE(myRecord) := ; //empty expression = cleared row
```

2. Se houver apenas um único padrão com um registro associado, e esse registro corresponder com o tipo de regra que está sendo definido. Por exemplo:

```
RULE(myRecord) e0 := '(' USE(myRecord, 'expression') ')';
```

Definições de ParsePattern

Um *parsepattern* pode conter qualquer combinação dos seguintes elementos:

<i>pattern-name</i>	O nome de qualquer atributo PATTERN previamente definido.
(<i>pattern</i>)	Parênteses podem ser usados para agrupamento.

Referência a Linguagem ECL
Suporte a Análise (Parsing)

<i>pattern1 pattern2</i>	<i>Padrão 1</i> acompanhamento por <i>padrão 2</i> .
'string'	Uma <i>string</i> de texto fixo que pode conter caracteres de controle de escape da string octal (por exemplo, CtrlZ é '\032').
FIRST	Corresponde com o início da string a ser buscada. Isto é semelhante a expressão regular ^ token, que <u>não</u> é suportada.
LAST	Corresponde com o final da string a ser buscada. Isto é semelhante a expressão regular \$ token, que <u>não</u> é suportada.
ANY	Corresponde com qualquer caractere.
REPEAT (<i>pattern</i>)	Repete o <i>padrão</i> em qualquer número de vezes. O <i>padrão*</i> de sintaxe da expressão regular é suportado como um atalho para REPEAT(<i>padrão</i>).
REPEAT (<i>pattern</i> , <i>expression</i>)	Repita o número de vezes da <i>expressão do padrão</i> . O <i>padrãotempo*</i> de sintaxe da <i>expressão regular</i> <count> é suportado como um atalho para REPEAT, mas a <i>expressão regular delimitada que repete o padrão</i> de sintaxe {expressão} <u>Não</u> é.
REPEAT (<i>pattern</i> , <i>low</i> , ANY [,MIN])	Repita o número de vezes do <i>padrão para</i> baixo ou mais vezes (com a opção MIN tornando-o uma correspondência mínima) O padrão + de sintaxe da expressão regular é suportado como um atalho para REPEAT(<i>padrão</i> , <i>baixo</i> , ANY), mas a <i>expressão regular delimitada que repete o padrão de sintaxe</i> {expressão} <u>não</u> é.
REPEAT (<i>pattern</i> , <i>low</i> , <i>high</i>)	Repita o número de vezes do <i>padrão de baixo para alto</i> Repita o número de vezes do padrão { <i>low</i> , <i>high</i> }. A expressão regular delimitada que repete o padrão { <i>low</i> , <i>high</i> } de sintaxe <u>não</u> é suportada.
OPT (<i>pattern</i>)	Um <i>padrão</i> opcional. O <i>padrão?</i> de sintaxe da expressão regular é suportado como um atalho para OPT (<i>pattern</i>).
<i>pattern1 OR pattern2</i>	Tanto <i>pattern1</i> quanto <i>pattern2</i> . O <i>padrão1 padrão2</i> de sintaxe da expressão regular é suportado como um atalho para OR.
[<i>list-of-patterns</i>]	Uma lista delimitada por vírgula dos <i>padrões</i> alternativos úteis para os conjuntos de string. Este é o mesmo que OR.
<i>pattern1</i> [NOT] IN <i>pattern2</i>	O texto correspondente ao <i>padrão1</i> também corresponde com o <i>padrão2</i> ? <i>Pattern1</i> [NOT] = <i>pattern2</i> and <i>pattern1</i> != <i>pattern2</i> são iguais ao IN, porém seu uso faz mais sentido em algumas situações.
<i>pattern1</i> [NOT] BEFORE <i>pattern2</i>	Verifique se o <i>padrão2</i> especificado [não] precede o <i>padrão1</i> . O <i>padrão2</i> não é consumido a partir da entrada.
<i>pattern1</i> [NOT] AFTER <i>pattern2</i>	Verifique se o <i>padrão2</i> especificado [não] precede o <i>padrão1</i> . O <i>Padrão2</i> não consome nenhuma entrada. Também deve ser de comprimento fixo.
<i>pattern</i> LENGTH(<i>range</i>)	Verifique se o comprimento do <i>padrão</i> está dentro do <i>intervalo</i> . O <i>intervalo</i> pode ter a forma <value>,<min>.. <i>max</i> >,<min>.. <i>max</i> Assim, "digit*3 NOT BEFORE digit" poderia ser representado como "digit* LENGTH(3)." Isto é mais eficiente, e digit* pode ser definido como um token. "digit* LENGTH(4..6)" corresponde com as sequências de dígito 4,5 e 6.
VALIDATE (<i>pattern</i> , <i>isValidExpression</i>)	Avalie <i>isValidExpression</i> para verificar se o <i>padrão</i> é ou não é válido. <i>isValidExpression</i> deve usar MATCHTEXT ou MATCHUNICODE para se referir ao texto correspondente ao <i>padrão</i> . Por exemplo, VALIDATE(alpha*, MATCHTEXT[4]='Q') é equivalente a alpha* = ANY*3 'Q' ANY* ou de forma mais útil : VALIDATE(alpha*,isSurnameService(MATCHTEXT));
VALIDATE (<i>pattern</i> , <i>isValidAsciiExpression</i> , <i>isValidUnicodeExpression</i>)	Uma variante de dois parâmetros. Use o primeiro <i>isValidAsciiExpression</i> se a string que estiver sendo pesquisada for ASCII; use o segundo se a string for Unicode.

Referência a Linguagem ECL
Suporte a Análise (Parsing)

NOCASE (<i>pattern</i>)	Corresponde com o <i>padrão</i> sem distinção entre maiúsculas e minúsculas, substituindo a opção CASE na função PARSE . Isto pode ser aninhado dentro de um padrão da opção CASE.
CASE (<i>pattern</i>)	Correspondências <i>pattern</i> corresponde com o padrão com distinção entre maiúsculas e minúsculas, substituindo a opção NOCASE na função PARSE . Isto pode ser aninhado dentro de um padrão da opção NOCASE .
<i>pattern</i> PENALTY (<i>cost</i>)	Associe um <i>custo</i> de penalidade à essa correspondência do <i>padrão</i> . Isto pode ser usado para recuperar de gramáticas com palavras desconhecidas. Isto requer o uso da opção BEST na operação PARSE.
TOKEN (<i>pattern</i>)	Trate o <i>padrão</i> como um token.
PATTERN ('regular expression')	<p>Define um padrão usando a <i>expressão regular</i> constituída a partir dos seguintes elementos de sintaxe suportados:</p> <p>(x) Agrupamento (não deve ser usado para correspondência)</p> <p>x y Alternativos x ou y xy</p> <p>xy Concatenação de x e y.</p> <p>x* x*? Zero ou mais. Greedy e versões mínimas.</p> <p>x+ x+? Um ou mais. Greedy e versões mínimas.</p> <p>x? x?? Zero ou um. Greedy e versões mínimas</p> <p>x{m} x{m,} x{m,n} Repetições delimitadas, também versões mínimas</p> <p>[0-9abcdef] Um conjunto de caracteres (pode usar ^ para a lista de exclusão)</p> <p>(?=...) (?!...) Declaração look-ahead</p> <p>(?<=...) (?<!...) Declaração look-behind</p> <p>Sequências Escape de declaração look-behind podem ser usadas para definir intervalos de caracteres UNICODE. A codificação é Big Endian UTF-16. Por exemplo: PATTERN AnyChar := PATTERN(U'[\u0001-\u7fff]');</p>
	<p>As seguintes expressões de classes de caractere são suportadas (dentro dos conjuntos):</p> <p>[:alnum:] [:cntrl:] [:lower:] [:upper:] [:space:]</p> <p>[:alpha:] [:digit:] [:print:] [:blank:] [:graph:]</p> <p>[:punct:] [:xdigit:]</p>
	<p><i>Expressões regulares</i> não suportam</p> <p>^ \$ para marcar o início/fim dos símbolos de agrupamento da string [.ch.]</p> <p>Equivalence class [=e=]</p>
USE ([<i>restruct</i> ,] 'symbol-name')	Especifica o uso de um padrão definido posteriormente com a função DEFINE('symbolname') . 'symbolname'function, Isso cria uma , prática apenas nos padrões RULE para análise tomata (a opção PARSE está presente na função PARSE).
SELF	Referencia o padrão que está sendo definido (recursivo). Isto é prático apenas nos padrões RULE para análise tomata (a opção PARSE está presente na função PARSE).

Exemplos:

```
rs := RECORD
STRING100 line;
END;
ds := DATASET([{'the fox; and the hen'}], rs);
```

```
PATTERN ws := PATTERN('[ \t\r\n]');
PATTERN Alpha := PATTERN('[A-Za-z]');
PATTERN Word := Alpha+;
PATTERN Article := ['the', 'A'];
PATTERN JustAWord := Word PENALTY(1);
PATTERN notHen := VALIDATE(Word, MATCHTEXT != 'hen');
PATTERN NoHenWord := notHen PENALTY(1);
RULE NounPhraseComponent1 := JustAWord | Article ws Word;
RULE NounPhraseComponent2 := NoHenWord | Article ws Word;
ps1 := RECORD
    out1 := MATCHTEXT(NounPhraseComponent1);
END;

ps2 := RECORD
    out2 := MATCHTEXT(NounPhraseComponent2);
END;

p1 := PARSE(ds, line, NounPhraseComponent1, ps1, BEST, MANY, NOCASE);
p2 := PARSE(ds, line, NounPhraseComponent2, ps2, BEST, MANY, NOCASE);
OUTPUT(p1);
OUTPUT(p2);
```

Ver também: PARSE, Estrutura RECORD, Estrutura TRANSFORM, DATASET

Funções NLP RECORD e TRANSFORM

As funções a seguir são usadas em expressões de definição de campo dentro da estrutura *RECORD* ou da função *TRANSFORM* que define o conjunto de resultado a partir da função *PARSE* :

MATCHED([*patternreference*])

MATCHED retorna “true” (verdadeiro) ou “false” (falso) quando *patternreference* encontra (ou não encontra) uma correspondência. Se *patternreference* for omitido, **MATCHED** indica se o padrão inteiro foi ou não foi combinado (usar com a opção **NOT MATCHED**).

MATCHTEXT [(*patternreference*)]

MATCHTEXT retorna o texto ASCII correspondente localizado por *patternreference* , ou se a correspondência não for encontrada. Se *patternreference* for omitido, **MATCHTEXT** retorna todos os textos correspondentes.

MATCHUNICODE(*patternreference*)

MATCHUNICODE retorna o texto Unicode correspondente localizado por *patternreference* , ou se a correspondência não for encontrada, **MATCHUNICODE** retornará em branco.

MATCHLENGTH(*patternreference*)

MATCHLENGTH retorna o número de caracteres no texto correspondente localizado por *patternreference* , ou se a correspondência não for encontrada, **MATCHLENGTH** retornará o número 0.

MATCHPOSITION(*patternreference*)

MATCHPOSITION retorna a posição do primeiro caractere dentro do texto no texto correspondente localizado por *patternreference* , ou se a correspondência não for encontrada, **MATCHPOSITION** retornará o número 0.

MATCHROW(*patternreference*)

MATCHROW retorna a linha inteira do texto correspondente localizado por *patternreference* para *RULE* (válido apenas quando a opção *PARSE* for usada na função *PARSE*). Isto pode ser usado para a qualificação completa de um campo na estrutura *RECORD* da linha.

Referências do Parâmetro Pattern

O parâmetro *patternreference* destas funções é uma lista delimitada por barra (/) dos atributos *PATTERN*, *TOKEN*, ou *RULE* previamente definidos, com ou sem um número de instância anexado em colchetes.

Se um número de instância for fornecido, o *patternreference* fará a correspondência de uma ocorrência específica; caso contrário, a correspondência será feita em qualquer ocorrência. O *patternreference* fornece um caminho através da gramática da expressão regular para um determinado resultado. O caminho para um atributo específico pode ser totalmente ou parcialmente especificado.

Exemplo:

```
PATTERN ws := PATTERN('[ \t\r\n]');
PATTERN arb := PATTERN('[!.,\t a-zA-Z0-9]')+;
PATTERN number := PATTERN('[0-9]')+;
PATTERN age := '(' number OPT('/I') ')';
PATTERN role := '[' arb ']';
PATTERN m_rank := '<' number '>';
PATTERN actor := arb OPT(ws '(I)' ws);
```

```
NLP_layout_actor_movie := RECORD
  STRING30 actor_name := MATCHTEXT(actor);
  STRING50 movie_name := MATCHTEXT(arb[2]); //2nd instance of arb
  UNSIGNED2 movie_year := (UNSIGNED)MATCHTEXT(age/number);
                        //number within age
  STRING20 movie_role := MATCHTEXT(role/arb); //arb within role
  UNSIGNED1 cast_rank := (UNSIGNED)MATCHTEXT(m_rank/number);
END;

// This example demonstrates the use of productions in PARSE code
//(only supported in the tomita version of PARSE).
PATTERN ws := [' ','\t'];
TOKEN number := PATTERN('[0-9]+');
TOKEN plus := '+';
TOKEN minus := '-';

attrRec := RECORD
  INTEGER val;
END;

RULE(attrRec) e0 :=
  '(' USE(attrRec,expr)? ')' |
  number TRANSFORM(attrRec, SELF.val := (INTEGER)$1;) |
  '-' SELF TRANSFORM(attrRec, SELF.val := -$2.val);
RULE(attrRec) e1 :=
  e0 |
  SELF '*' e0 TRANSFORM(attrRec, SELF.val := $1.val * $3.val;) |
  USE(attrRec, e1) '/' e0
    TRANSFORM(attrRec, SELF.val := $1.val / $3.val);
RULE(attrRec) e2 :=
  e1 |
  SELF plus e1 TRANSFORM(attrRec, SELF.val := $1.val + $3.val;) |
  SELF minus e1 TRANSFORM(attrRec, SELF.val := $1.val - $3.val);
RULE(attrRec) expr := e2;

infile := DATASET([{'1+2*3'},{'1+2*z'},{'1+2+(3+4)*4/2'}],
  { STRING line });
resultsRec := RECORD
  RECORDOF(infile);
  attrRec;
  STRING exprText;
  INTEGER value3;
END;

resultsRec extractResults(infile l, attrRec attr) := TRANSFORM
  SELF := l;
  SELF := attr;
  SELF.exprText := MATCHTEXT;
  SELF.value3 := MATCHROW(e0[3]).val;
END;

OUTPUT(PARSE(infile,line,expr,extractResults(LEFT, $1),
  FIRST,WHOLE,PARSE,SKIP(ws)));
```

Ver também: PARSE, Estrutura RECORD, Estrutura TRANSFORM

Funções RECORD e TRANSFORM e para Análise de XML

As seguintes funções são válidas para uso apenas em expressões de definição de campo dentro de uma estrutura *RECORD* ou de uma função *TRANSFORM* usada para definir o conjunto de resultado da função *PARSE*, ou a estrutura *RECORD* de entrada para um *DATASET* que contém dados XML.

XMLTEXT(*xmltag*)

XMLTEXT returns the ASCII text from the *xmltag*.

XMLUNICODE(*xmltag*)

XMLUNICODE retorna o texto Unicode da *xmltag*.

XMLPROJECT(*xmltag, transform*)

XMLPROJECT retorna o texto da *xmltag* na forma de um dataset secundário.

<i>xmltag</i>	Uma constante de string que nomeia o XPATH para a tag que contém os dados (consulte a seção Suporte XPATH na discussão sobre a estrutura <i>RECORD</i>). Pode conter um número de instância (tal como <i>tagname[1]</i>).
<i>transform</i>	A função <i>TRANSFORM</i> que gera o dataset secundário.

Exemplo:

```
d := DATASET([{'<library><book isbn="123456789X">' +
  '<author>Bayliss</author><title>A Way Too Far</title></book>' +
  '<book isbn="1234567801">' +
  '<author>Smith</author><title>A Way Too Short</title></book>' +
  '</library>'}],
{STRING line });

rform := RECORD
  STRING author := XMLTEXT('author');
  STRING title := XMLTEXT('title');
END;

books := PARSE(d,line,rform,XML('library/book'));

OUTPUT(books);

/*****
/* The following XML can be parsed using XMLPROJECT
<XML>
<Field name='surname' distinct=2>
<Value count=3>Halliday</Value>
<Value count=2>Chapman</Value>
</Field>
<XML>
*/
extractedValueRec := RECORD
  STRING value;
  UNSIGNED cnt;
END;

extractedRec := RECORD
  STRING name;
```

```
    UNSIGNED cnt;
    DATASET(extractedValueRec) values;
END;

x := DATASET([{'<XML>' +
              '<Field name="surname" distinct="2">' +
              '<Value count="3">Halliday</Value>' +
              '<Value count="2">Chapman</Value>' +
              '</Field>' +
              '</XML>'}], {STRING line});

extractedRec t1 := TRANSFORM
  SELF.name    := XMLTEXT('@name');
  SELF.cnt     := (UNSIGNED)XMLTEXT('@distinct');
  SELF.values := XMLPROJECT('Value',
                           TRANSFORM(extractedValueRec,
                                       SELF.value := XMLTEXT(''),
                                       SELF.cnt :=
                                         (UNSIGNED)XMLTEXT('@count'))(cnt > 1);
END;
p := PARSE(x, line, t1, XML('XML/Field'));
OUTPUT(p);
```

Ver também: PARSE, Estrutura RECORD, Estrutura TRANSFORM, DATASET

Palavras-chave reservadas

ALL

ALL

A **palavra-chave** ALL especifica o conjunto de todos os valores possíveis quando usados como o valor padrão de um parâmetro SET especificado, ou como substituição de um SET em operações que exigem um SET (CONJUNTO) definido de valores.

Exemplo:

```
MyFunc(String1 val, SET OF String1 S=ALL) := val IN S;  
    //check for presence in passed set, if passed  
  
SET OF Integer4 MySet := IF(SomeCondition=TRUE,  
    [88888,99999,66666,33333,55555],ALL);  
MyRecs := MyFile(Zip IN MySet);
```

Ver também: SET OF, Funções dos atributos (Especificações de parâmetros)

EXCEPT

EXCEPT *fieldlist*

fields Uma lista delimitada por vírgula dos campos de dados em uma estrutura RECORD.

The **EXCEPT** keyword specifies a list of *fields* not to use in a SORT, GROUP, DEDUP, or ROLLUP operation. Isso permite a realização da operação em todos os campos no RECORD, EXCEPT naqueles *campos* que você nomear, tornando o código mais legível e passível de manutenção.

Exemplo:

```
x := DATASET([{'Taylor','Richard','Jackson' , 'M'},
              {'Taylor','David' , 'Boca' , 'M'},
              {'Taylor','Rita' , 'Boca' , 'F'},
              {'Smith' , 'Richard','Mansfield','M'},
              {'Smith' , 'Oscar' , 'Boca' , 'M'},
              {'Smith' , 'Rita' , 'Boca' , 'F'}],
             {STRING10 lname, STRING10 fname,
              STRING10 city, STRING1 sex });
y := SORT(x, EXCEPT sex); //sort on all fields but sex

OUTPUT(y)
```

Ver também: SORT, GROUP, DEDUP, ROLLUP

EXPORT

EXPORT [VIRTUAL] *definition*

VIRTUAL	Opcional. Especifica que a definição é <i>VIRTUAL</i> Válido apenas dentro de uma estrutura <i>MODULE</i> .
<i>definition</i>	Uma definição válida

A palavra-chave **EXPORT** permite explicitamente que outras definições importem a *definição* especificada que será usada. Ela deve ser *IMPORTed* a partir do código de qualquer pasta; por isso, seu escopo de visibilidade é global.

O código ECL é armazenado em arquivos de texto .ecl, que devem conter apenas uma única definição **EXPORT** ou **SHARED**. Essa definição pode ser uma estrutura que permite as definições **EXPORT** ou **SHARED** dentro de suas áreas de acesso (tais como *MODULE*, *INTERFACE*, *TYPE*, etc.). O nome do arquivo .ecl que contém o código deve ser exatamente o mesmo que a definição **EXPORT** (ou **SHARED**) nele contida.

Definições sem as palavras-chave **EXPORT** ou **SHARED** são consideradas locais em relação ao arquivo no qual residem (ver Visibilidade das definições). O escopo de uma *definição* local está limitado à próxima definição **SHARED** ou **EXPORT**; desta forma, ele deve preceder a definição **EXPORT** ou **SHARED** desse arquivo.

Exemplo:

```
EXPORT MyDefinition := 5;
// allows other definitions to use MyModule.MyDefinition if they import MyModule
// the filename must be MyDefinition.ecl

//and in AnotherDef.ecl we have this code:
EXPORT AnotherDef := MODULE(x)
  EXPORT INTEGER a := c * 3;
  EXPORT INTEGER b := 2;
  EXPORT VIRTUAL INTEGER c := 3; //this def is VIRTUAL
END;
```

Ver também: **IMPORT**, **SHARED**, Visibilidade das definições, Estrutura *MODULE*

Palavra-chave GROUP

GROUP

A palavra-chave **GROUP** é usada dentro do parâmetro de resultado *format* (estrutura RECORD) da definição TABLE, onde grupos opcionais por *expressões* também estão presentes. GROUP substitui o parâmetro *recordset* de qualquer função embutida agregada usada no resultado, para indicar que a operação é desempenhada por cada grupo da *expressão*. Isso é semelhante à cláusula “GROUP BY” do SQL . O uso mais comum é gerar uma tabela na forma de um relatório de tabela de referência cruzada.

Há também uma função GROUP embutida que oferece funcionalidade semelhante.

Exemplo:

```
A := TABLE(Person, {per_st, per_sex, COUNT(GROUP)}, per_st, per_sex);  
// create a crosstab report of each sex in each state
```

Ver também: TABLE, COUNT, AVE, MAX, MIN, SUM, VARIANCE, COVARIANCE, CORRELATION, COMBINE

IMPORT

IMPORT *module-selector-list*;

IMPORT *folder* **AS** *alias* ;

IMPORT *symbol-list* **FROM** *folder* ;

IMPORT*language*;

<i>module-selector-list</i>	Uma lista delimitada por vírgula de pastas ou nomes de arquivos no repositório. O símbolo monetário (\$) torna disponível todas as definições na pasta atual. O acento circunflexo (^) pode ser usado como abreviação do contêiner da pasta atual. O uso do acento circunflexo no especificador de módulo (como por exemplo, myModule.^) seleciona o contêiner dessa pasta. O acento circunflexo principal especifica a raiz lógica da árvore do arquivo.
<i>folder</i>	Nome da pasta ou do arquivo no repositório.
AS	Define um nome de <i>alias</i> local para a <i>pasta</i> , sendo normalmente usado para criar nomes locais mais curtos para facilitar a digitação.
<i>alias</i>	O nome abreviado a ser usado em vez do nome da <i>pasta</i> .
<i>symbol-list</i>	Uma lista, delimitada por vírgula, de definições da <i>pasta</i> a ser disponibilizada sem qualificação. Um único asterisco (*) deve ser usado para disponibilizar todas as definições da <i>pasta</i> sem a necessidade de qualificação.
FROM	Especifica o nome da <i>pasta</i> na qual a <i>symbol-list</i> reside.
<i>language</i>	Especifica o nome de uma linguagem de programação externa cujo código você deseja incorporar à sua ECL. O módulo de suporte à linguagem – para a linguagem que deseja incorporar – precisa estar instalado em seu diretório de plugins. Isso faz com que a <i>linguagem</i> possa ser usada pela estrutura EMBED e/ou pela função IMPORT.

A palavra-chave **IMPORT** disponibiliza as definições EXPORT (e SHARED da mesma *pasta*) para serem usadas no código ECL atual.

Exemplos:

```
IMPORT $; //makes all definitions from the same folder available

IMPORT $, Std; //makes the standard library functions available, also

IMPORT MyModule; //makes available the definitions from MyModule folder

IMPORT $.^MyOtherModule //makes available the definitions from MyOtherModule folder,
//located in the same container as the current folder

IMPORT $.^.^SomeOtherModule //makes available the definitions from SomeOtherModule folder,
//which is located in the grandparent folder of current folder

IMPORT SomeFolder.SomeFile; //make the specific file available

IMPORT SomeReallyLongFolderName AS SN; //alias the long name as "SN"

IMPORT ^ as root; //allows access to non-modules defined
//in the root of the repository

IMPORT Def1,Def2 FROM Fred; //makes Def1 and Def2 from Fred folder available, unqualified
```

Referência a Linguagem ECL

Palavras-chave reservadas

```
IMPORT * FROM Fred;           //makes everything from Fred available, unqualified
IMPORT Dev.Me.Project1;       //makes the Dev/Me/Project1 folder available
IMPORT Python;                 //makes Python language code embeddable
```

Ver também: EXPORT, SHARED, Estrutura EMBED, Função IMPORT

KEYED e WILD

KEYED(*expression* [, **OPT**])

WILD(*field*)

<i>expression</i>	Uma condição de filtro INDEX.
OPT	Gera apenas uma condição de filtro.
<i>field</i>	Um único campo em um INDEX.

As palavras-chave **KEYED WILD** são válidas apenas para filtros nos atributos INDEX (que também qualifica como parte da *junção* JOIN com "half-keyed"). Elas indicam ao compilador quais dos principais campos de índice são usados como filtros (KEYED) ou como curinga (WILD) para que o compilador possa avisá-lo em caso de erro. Os campos de seguimento não usados no filtro são ignorados (sempre tratados como curingas).

As regras de uso são as seguintes (o termo “segmonitor” se refere a um objeto interno criado para representar possíveis condições de coincidência para um campo de chave única):

1. KEYED gera um segmonitor. O segmonitor pode ser wild se a expressão nunca puder ser falsa, tal como: *expression* o segmonitor pode ser wild se a expressão nunca puder ser falsa, tal como:

```
KEYED(inputval = '' OR field = inputval)
```

2. WILD gera um segmonitor wild, a menos que também exista um filtro KEYED() no mesmo campo.

3. KEYED, OPT gera um segmonitor não wild apenas se o campo anterior também o fez.

4. Qualquer campo que contenha KEYED e KEYED OPT gera erro no tempo de compilação.

5. Se WILD ou KEYED não forem especificados para nenhum campo, os segmonitores são gerados para todas as condições inseridas.

6. Uma condição de filtro INDEX sem o KEYED especificado, gera um segmonitor wild (exceto como especificado no item 5).

7. Os limites KEYED baseiam-se em todos os segmonitores não wild.

8. As condições que não geram segmonitores são pós-filtradas.

Exemplo:

```
ds := DATASET('~local::rkc::person',
  { STRING15 f1, STRING15 f2, STRING15 f3, STRING15 f4,
    UNSIGNED8 filepos{VIRTUAL(fileposition)} }, FLAT);
ix := INDEX(ds, { ds }, '\\lexis\\person.name_first.key');

/**** Valid examples ****/

COUNT(ix(KEYED(f1='Kevin1')));
// legal because only f1 is used.

COUNT(ix(KEYED(f1='Kevin2' and f2='Halliday')));
// legal because both f1 and f2 are used

COUNT(ix(KEYED(f2='Kevin3') and WILD(f1)));
// keyed f2, but ok because f1 is marked as wild.
```

```
COUNT(ix(f2='Halliday'));
    // ok - if keyed isn't used then it doesn't have to have
    // a wild on f1

COUNT(ix(KEYED(f1='Kevin3') and KEYED(f2='Kevin4') and WILD(f1)));
    // it is ok to mark as wild and keyed otherwise you can get
    // in a mess with compound queries.

COUNT(ix(f1='Kevin3' and KEYED(f2='Kevin4') and WILD(f1)));
    // can also be wild and a general expression.

/**Error examples **/

COUNT(ix(KEYED(f3='Kevin3' and f2='Halliday')));
    // missing WILD(f1) before keyed

COUNT(ix(KEYED(f3='Kevin3') and f2='Halliday'));
    // missing WILD(f1) before keyed after valid field

COUNT(ix(KEYED(f3='Kevin3') and WILD(f2)));
    // missing WILD(f1) before a wild

COUNT(ix(WILD(f3) and f2='Halliday'));
    // missing WILD(f1) before wild after valid field

COUNT(ds(KEYED(f1='Kevin')));
    //KEYED not valid in DATASET filters
```

Ver também: INDEX, JOIN, FETCH

LEFT e RIGHT

LEFT

RIGHT

A **LEFT** e **RIGHT** as palavras-chave LEFT e RIGHT indicam os registros esquerdo e direito de um conjunto de registros. Elas podem ser usadas para substituir na forma de parâmetros especificados às funções TRANSFORM ou em expressões nas funções onde os registros esquerdo e direito estejam implícitos, como por exemplo DEDUP e JOIN.

Exemplo:

```
dup_flags := JOIN(person, person,  
    LEFT.current_address_key=RIGHT.current_address_key  
    AND fuzzy_equal, req_output(LEFT, RIGHT));
```

Ver também: Estrutura TRANSFORM, DEDUP

LIKELY e UNLIKELY

[*attrname* :=] **LIKELY**(*filtercondition*, [*likelihood*]);

[*attrname* :=] **UNLIKELY**(*filtercondition*);

<i>filtercondition</i>	Uma condição de filtro para a dica.
<i>likelihood</i>	O valor da probabilidade expresso em número decimal entre 0 e 1.

A dica **LIKELY**/**UNLIKELY** pode ser envolvida em torno de uma condição de filtro para indicar ao gerador de código a probabilidade de a condição de filtro filtrar o registro.

LIKELY especifica que a condição de filtro provavelmente coincidirá com a maioria dos registros. **UNLIKELY** especifica que apenas alguns poucos registros provavelmente coincidirão.

Um valor específico de probabilidade deve ser fornecido para **LIKELY**. O valor de probabilidade consiste de um número decimal maior do que 0 e menor do que 1. Quanto mais próximo estiver de 1.0, maior será a probabilidade de a condição de filtro coincidir com o registro. Quanto mais próximo estiver de 0.0, menor será a probabilidade de a condição de filtro coincidir com os registros. O gerador de código utiliza as informações de probabilidade para gerar um código melhor.

O gerador de código usa a dica **LIKELY**/**UNLIKELY** juntamente com a contagem de uso determinar o custo do despejo e o custo de uma nova filtragem do dataset toda vez que ele for usado. Os despejos são gerados apenas quando o seu custo é menor do que o custo de uma nova filtragem do dataset.

Por exemplo, digamos que existe um dataset de pessoas com milhares de registros. Cria-se um filtro para reter todos os registros cuja idade dessas pessoas seja abaixo de 100 anos. O filtro provavelmente reterá 99,9% dos registros. O resultado do filtro é usado por 3 atividades distintas. O custo para despejar os resultados do filtro provavelmente será muito maior do que o de uma nova filtragem do dataset de entrada toda vez que for usado. **LIKELY** pode ser usado para compartilhar essa informação de probabilidade com o gerador de código para que ele tome decisões sensatas sobre quando o despejo deve ser feito.

Exemplo:

```
PeopleYoungerThan100 := AllPeople( LIKELY(age < 100, 0.999) );  
// Probably not worth spilling PeopleYoungerThan100  
  
PeopleOlderThan100 := AllPeople( UNLIKELY(age>100) );  
// Probably worth spilling even if PeopleOlderThan100 is used by only a couple of activities
```

ROWS(LEFT) e ROWS(RIGHT)

ROWS(LEFT)

ROWS(RIGHT)

As palavras-chave **ROWS(LEFT)** e **ROWS(RIGHT)** indicam que o parâmetro está sendo especificado para a função TRANSFORM é um conjunto de registros. Elas são usadas em funções onde um dataset está sendo especificado, tais como COMBINE, ROLLUP, JOIN, DENORMALIZE, e LOOP.

Exemplo:

```
NormRec := RECORD
  STRING20 thename;
  STRING20 addr;
END;
NamesRec := RECORD
  UNSIGNED1 numRows;
  STRING20 thename;
  DATASET(NormRec) addresses;
END;
NamesTable := DATASET([ {0,'Kevin',[ ]},{0,'Liz',[ ]},
                        {0,'Mr Nobody',[ ]},{0,'Anywhere',[ ]}],
                      NamesRec);
NormAddrs := DATASET([ {'Kevin','10 Malt Lane'},
                        {'Liz','10 Malt Lane'},
                        {'Liz','3 The cottages'},
                        {'Anywhere','Here'},
                        {'Anywhere','There'},
                        {'Anywhere','Near'},
                        {'Anywhere','Far'}],NormRec);
NamesRec DeNormThem(NamesRec L, DATASET(NormRec) R) := TRANSFORM
  SELF.NumRows := COUNT(R);
  SELF.addresses := R;
  SELF := L;
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
                             LEFT.thename = RIGHT.thename,
                             GROUP,
                             DeNormThem(LEFT,ROWS(RIGHT)));
OUTPUT(DeNormedRecs);
```

Ver também: Estrutura TRANSFORM, COMBINE, ROLLUP , JOIN, DENORMALIZE, LOOP

SELF

SELF.*element*

element O nome de um campo no tipo de resultado da estrutura RECORD de uma estrutura TRANSFORM .

A **palavra-chave** SELF é usada nas estruturas TRANSFORM para indicar um campo na estrutura de resultado. Ela não deve ser usada ao lado direito de qualquer definição de atributo.

Exemplo:

```
Ages := RECORD
    INTEGER8 Age; //a field named "Age"
END;

TodaysYear := 2001;
Ages req_output(person l) := TRANSFORM
    SELF.Age := TodaysYear - l.birthdate[1..4];
END;
```

Ver também: Estrutura TRANSFORM

SHARED

SHARED [VIRTUAL] *definition*

VIRTUAL	Opcional. Especifica que a definição é <i>VIRTUAL</i> Válido apenas dentro de uma estrutura MODULE .
<i>definition</i>	Uma definição válida

A palavra-chave **SHARED** permite, de forma explícita, que outras definições dentro de uma mesma pasta importem a *definição* especificada para ser usada em todo o módulo/pasta/diretório (p.ex., escopo do módulo), mas não fora desse escopo.

O código ECL é armazenado em arquivos de texto .ecl, que devem conter apenas uma única definição **EXPORT** ou **SHARED**. Essa definição pode ser uma estrutura que permite as definições **EXPORT** ou **SHARED** dentro de suas áreas de acesso (tais como **MODULE**, **INTERFACE**, **TYPE**, etc.). O nome do arquivo .ecl que contém o código deve ser exatamente o mesmo que a definição **EXPORT** (ou **SHARED**) nele contida.

Definições sem as palavras-chave **EXPORT** ou **SHARED** são consideradas locais em relação ao arquivo no qual residem (ver Visibilidade das definições). O escopo de uma *definição* local está limitado à próxima definição **SHARED** ou **EXPORT**; desta forma, ele deve preceder a definição **EXPORT** ou **SHARED** desse arquivo.

Exemplo:

```
//this code is contained in the GoodHouses.ecl file
BadPeople := Person(EXISTS(trades(EXISTS(phr(phr_rate > '4'))));
    //local only to the GoodHouses definition
SHARED GoodHouses := Household(~EXISTS(BadPeople));
    //available all thru the module

//and in AnotherDef.ecl we have this code:
EXPORT AnotherDef := MODULE(x)
    EXPORT INTEGER a := c * 3;
    EXPORT INTEGER b := 2;
    SHARED VIRTUAL INTEGER c := 3; //this def is VIRTUAL
    EXPORT VIRTUAL INTEGER d := c + 3; //this def is VIRTUAL
    EXPORT VIRTUAL INTEGER e := c + 3; //this def is VIRTUAL
END;
```

Ver também: **IMPORT**, **EXPORT**, Visibilidade das definições, Estrutura **MODULE**

SKIP

SKIP

SKIP só é válido dentro de uma estrutura TRANSFORM e pode ser usada em qualquer caso que lugar em que uma expressão é aplicada para indicar que o registro de resultado atual não deve ser gerado no conjunto de resultados. COUNTER COUNTER são incrementados mesmo quando SKIP elimina a criação do registro atual.

Exemplo:

```
SequencedAges := RECORD
    Ages;
    INTEGER8 Sequence := 0;
END;

SequencedAges AddSequence(Ages l, INTEGER c) := TRANSFORM
    SELF.Sequence := IF(c % 2 = 0, SKIP,c); //skip the even recs
    SELF := l;
END;

SequencedAgedRecs := PROJECT(AgedRecs, AddSequence(LEFT,COUNTER));
```

Ver também: Estrutura TRANSFORM

TRUE e FALSE

TRUE

FALSE

As palavras-chave **TRUE** e **FALSE** são constantes booleanas.

Exemplo:

```
BooleanTrue := TRUE;  
Booleanfalse := FALSE;
```

Ver também: **BOOLEAN**

Estruturas Especiais

Estrutura BEGINC++

resulttype *funcname* (*parameterlist*) := **BEGINC++**

code

ENDC++;

<i>resulttype</i>	O tipo do valor de retorno da ECL para a função C++.
<i>funcname</i>	O nome da definição ECL da função.
<i>parameterlist</i>	Uma lista separada por vírgulas com os parâmetros a serem passados para a <i>função</i> .
<i>code</i>	O código fonte da função C++.

A estrutura **BEGINC++** possibilita a adição de código C++ em linha à sua ECL. Isso é útil quando o processamento de strings ou bits seria complicado na ECL e mais fácil em C++. Normalmente, é usado para código que será executado uma única vez. Para código C++ usado mais frequentemente, uma melhor solução seria criar um plugin (consulte a discussão **Implementação de serviços externos**).

A implementação deve ser codificada como segura para linha de execução (thread). Todas as chamadas para bibliotecas externas devem ser feitas para as versões dessas bibliotecas que são seguras para linha de execução.

Utilizando este form, você pode usar **EMBED** ao invés de **BEGINC++** para incorporar código C++ e especificar opções adicionais (por exemplo, **DISTRIBUTED**):

```
myFunction(string name) := EMBED(C++ [: options])
... text
ENDEMBED
```

ATENÇÃO: Esse recurso pode gerar corrupção de memória e/ou problemas de segurança. Portanto, recomendamos cautela e uma consideração detalhada. Consulte o Suporte Técnico antes de usar.

ECL para C++ Mapping

Os tipos são passados da seguinte forma:

```
//The following typedefs are used below:
typedef unsigned size32_t;
typedef wchar_t UChar; [ unsigned short in linux ]
```

A lista a seguir descreve os mapeamentos da ECL para o C++. Para C++ incorporado, os parâmetros são sempre convertidos em minúsculas, com maiúsculas nas conjunções (veja abaixo).

ECL	C++ [Linux in brackets]
BOOLEAN xyz	bool xyz
INTEGER1 xyz	signed char xyz
INTEGER2 xyz	int16_t xyz
INTEGER4 xyz	int32_t xyz
INTEGER8 xyz	signed __int64 xyz [long long]
UNSIGNED1 xyz	unsigned char xyz
UNSIGNED2 xyz	uint16_t xyz
UNSIGNED4 xyz	uint32_t xyz
UNSIGNED8 xyz	unsigned __int64 xyz [unsigned long long xyz]
REAL4 xyz	float xyz
REAL/REAL8 xyz	double xyz
DATA xyz	size32_t lenXyz, void * xyz
STRING xyz	size32_t lenXyz, char * xyz
VARSTRING xyz	char * xyz;
QSTRING xyz	size32_t lenXyz, char * xyz

Referência a Linguagem ECL

Estruturas Especiais

```
UNICODE xyz          size32_t lenXyz, UChar * xyz
VARUNICODE xyz        UChar * xyz
DATA<nn> xyz          void * xyz
STRING<nn> xyz        char * xyz
QSTRING<nn> xyz       char * xyz
UNICODE<nn> xyz       UChar * xyz
SET OF ... xyz        bool isAllXyz, size32_t lenXyz, void * xyz
```

Observe que strings de comprimento desconhecido são passadas de forma diferente das que têm tamanho conhecido. Uma string de comprimento variável é passada como um número de caracteres e não como tamanho (ou seja, qstring/unicode), seguido por um pointer para os dados, desta forma (size32_t é um UNSIGNED4):

```
STRING ABC -> size32_t lenAbc, const char * abc;
UNICODE ABC -> size32_t lenABC, const UChar * abc;
```

Um dataset é passado como um par tamanho/pointer. O comprimento determina o tamanho do seguinte dataset em bytes. A mesma convenção de nomenclatura é usada:

```
DATASET(r)           ABC -> size32_t lenAbc, const void * abc
  The rows are accessed as x+0, x + length(row1), x + length(row1) + length(row2)

LINKCOUNTED DATASET(r) ABC -> size32_t countAbc, const byte * * abc
  The rows are accessed as x[0], x[1], x[2]
```

OBSERVAÇÃO: strings de comprimento variável em um registro são armazenadas como um número de 4 bytes de caracteres, seguido pelos dados da string.

Os conjuntos são passados como um conjunto de parâmetros (all, size, pointer):

```
SET OF UNSIGNED4 ABC -> bool isAllAbc, size32_t lenAbc, const void * abc
```

Os tipos de retorno são processados como funções C++ retornando os mesmos tipos, com algumas exceções. As exceções têm alguns parâmetros iniciais adicionais onde os resultados serão retornados:

ECL	C++ [Linux in brackets]
DATA xyz	size32_t & __lenResult, void * & __result
STRING xyz	size32_t & __lenResult, char * & __result
CONST STRING xyz	size32_t lenXyz, const char * xyz
QSTRING xyz	size32_t & __lenResult, char * & __result
UNICODE xyz	size32_t & __lenResult, UChar * & __result
CONST UNICODE xyz	size32_t & __lenResult, const UChar * & __result
DATA<nn> xyz	void * __result
STRING<nn> xyz	char * __result
QSTRING<nn> xyz	char * __result
UNICODE<nn> xyz	UChar * __result
SET OF ... xyz	bool __isAllResult, size32_t & __lenResult, void * & __result
DATASET(r)	size32_t & __lenResult, void * & __result
LINKCOUNTED DATASET(r)	size32_t & __countResult, byte * * & __result
STREAMED DATASET(r)	returns a pointer to an IRowStream interface (see the eclhelper.hpp include file for the definition)

Por exemplo,

```
STRING process(STRING value, INTEGER4 len)
```

tem o protótipo:

```
void process(size32_t & __lenResult, char * & __result,
             size32_t lenValue, char * value, int len);
```

Uma função que recebe parâmetros de string também deve ter o tipo prefixado por **const** no código da ECL para que compiladores modernos não relatem erros quando strings constantes são passadas à função.

```
BOOLEAN isUpper(const string mystring) := BEGINC++
    size_t i=0;
    while (i < lenMystring)
    {
        if (!isupper((byte)mystring[i]))
            return false;
        i++;
    }
    return true;
ENDC++;
isUpper('JIM');
```

Opções disponíveis:

#option pure	Por padrão, supõe-se que as funções C++ incorporadas tenham efeitos colaterais. Isso significa que o código gerado não será tão eficiente como poderia, pois as chamadas não serão compartilhadas. A adição de #option dentro do <i>code</i> C++ incorporado faz como que ele seja tratado como uma função pura, sem efeitos colaterais.
#option once	Indica que a função não tem efeitos colaterais e é avaliada no tempo de execução da query, mesmo se os parâmetros forem constantes, o que permite que o otimizador faça chamadas mais eficientes à função em alguns casos
#option action	Indica efeitos colaterais, o que exige que o otimizador mantenha todas as chamadas à função.
#body	Delimita o início do código executável. Todo o <i>code</i> que precede #body (como #include) é gerado fora da definição da função. Todo o código subsequente é gerado dentro da definição da função.

Exemplo:

```
//static int add(int x,int y) {
INTEGER4 add(INTEGER4 x, INTEGER4 y) := BEGINC++
    #option pure
    return x + y;
ENDC++;

OUTPUT(add(10,20));

//static void reverseString(size32_t & __lenResult,char * & __result,
// size32_t lenValue,char * value) {
STRING reverseString(STRING value) := BEGINC++
    size32_t len = lenValue;
    char * out = (char *)rtlMalloc(len);
    for (unsigned i= 0; i < len; i++)
        out[i] = value[len-1-i];
    __lenResult = len;
    __result = out;
ENDC++;
OUTPUT(reverseString('Kevin'));
// This is a function returning an unknown length string via the
// special reference parameters __lenResult and __result

//this function demonstrates #body, allowing #include to be used
BOOLEAN nocaseInList(STRING search,
    SET OF STRING values) := BEGINC++
#include <string.h>
#body
```

```
if (isAllValues)
    return true;
const byte * cur = (const byte *)values;
const byte * end = cur + lenValues;
while (cur != end)
{
    unsigned len = *(unsigned *)cur;
    cur += sizeof(unsigned);
    if (lenSearch == len && memcmp(search, cur, len) == 0)
        return true;
    cur += len;
}
return false;
ENDC++;

//and another example, generating a variable number of Xes
STRING buildString(INTEGER4 value) := BEGINC++
    char * out = (char *)rtlMalloc(value);
    for (unsigned i= 0; i < value; i++)
        out[i] = 'X';
    __lenResult = value;
    __result = out;
ENDC++;

//examples of embedded, LINKCOUNTED, and STREAMED DATASETS
inRec := { unsigned id };
doneRec := { unsigned4 execid };
out1rec := { unsigned id; };
out2rec := { real id; };

DATASET(doneRec) doSomethingNasty(DATASET(inRec) input) := BEGINC++
    __lenResult = 4;
    __result = rtlMalloc(8);
    *(unsigned *)__result = 91823;
ENDC++;

DATASET(out1rec) extractResult1(doneRec done) := BEGINC++
    const unsigned id = *(unsigned *)done;
    const unsigned cnt = 10;
    __lenResult = cnt * sizeof(unsigned __int64);
    __result = rtlMalloc(__lenResult);
    for (unsigned i=0; i < cnt; i++)
        ((unsigned __int64 *)__result)[i] = id + i + 1;
ENDC++;

LINKCOUNTED DATASET(out2rec) extractResult2(doneRec done) := BEGINC++
    const unsigned id = *(unsigned *)done;
    const unsigned cnt = 10;
    __countResult = cnt;
    __result = _resultAllocator->createRowset(cnt);
    for (unsigned i=0; i < cnt; i++)
    {
        size32_t allocSize;
        void * row = _resultAllocator->createRow(allocSize);
        *(double *)row = id + i + 1;
        __result[i] = (byte *)__resultAllocator->finalizeRow(allocSize, row, allocSize);
    }
ENDC++;

STREAMED DATASET(out1rec) extractResult3(doneRec done) := BEGINC++
    class myStream : public IRowStream, public RtlCInterface
    {
    public:
        myStream(IEngineRowAllocator * _allocator, unsigned _id) :
            allocator(_allocator), id(_id), idx(0) {}
```

```
RTLIMPLEMENT_IINTERFACE

virtual const void *nextRow()
{
    if (idx >= 10)
        return NULL;
    size32_t allocSize;
    void * row = allocator->createRow(allocSize);
    *(unsigned __int64 *)row = id + ++idx;
    return allocator->finalizeRow(allocSize, row, allocSize);
}
virtual void stop() {}
private:
Linked<IEngineRowAllocator> allocator;
unsigned id;
unsigned idx;

};
#body
const unsigned id = *(unsigned *)done;
return new myStream(_resultAllocator, id);
ENDC++;

ds := DATASET([1,2,3,4], inRec);

processed := doSomethingNasty(ds);

out1 := NORMALIZE(processed, extractResult1(LEFT), TRANSFORM(RIGHT));
out2 := NORMALIZE(processed, extractResult2(LEFT), TRANSFORM(RIGHT));
out3 := NORMALIZE(processed, extractResult3(LEFT), TRANSFORM(RIGHT));

SEQUENTIAL(OUTPUT(out1),OUTPUT(out2),OUTPUT(out3));
```

Ver também: Implementação de Serviços Externos, Estrutura EMBED

Estrutura EMBED

resulttype funcname (parameterlist) := EMBED(language)

code

ENDEMBED;

resulttype funcname (parameterlist) := EMBED(language, code);

<i>resulttype</i>	O tipo do valor de retorno da função.
<i>funcname</i>	O nome da definição ECL da função.
<i>parameterlist</i>	Uma lista separada por vírgulas com os parâmetros a serem passados para a <i>função</i> .
<i>language</i>	O nome da linguagem de programação a ser incorporada. O módulo de suporte à linguagem – para a linguagem que deseja incorporar – precisa estar instalado em seu diretório de plugins. São fornecidos módulos para linguagens como Java, R, Javascript e Python. Você pode escrever o seu próprio módulo de suporte à linguagem plugável para qualquer linguagem que ainda não conta com suporte usando os módulos fornecidos como exemplos ou pontos de partida.
<i>code</i>	A código fonte a ser incorporado.

A estrutura **EMBED** possibilita a adição de código de *linguagem* em linha à sua ECL. Essa estrutura é semelhante à estrutura **BEGINC++**, mas está disponível para qualquer *linguagem* com um módulo de suporte à linguagem plugável instalado, como R, Javascript e Python. Outros módulos poderão ser fornecidos. Como alternativa, é possível escrever o seu próprio módulo usando os módulos fornecidos como modelos/exemplos/pontos de partida. Essa estrutura pode ser usada para escrever código Javascript, R ou Python, mas não para escrever código Java (use a função **IMPORT** para código Java).

Os tipos de parâmetro que podem ser passados e retornados variam em função de *linguagem*, mas de forma geral os tipos escalares simples (**INTEGER**, **REAL**, **STRING**, **UNICODE**, **BOOLEAN** e **DATA**), bem como os SETs desses tipos escalares, são permitidos desde que exista um tipo de dados adequado na *linguagem* que possa ser usado para mapear esses tipos escalares.

Esse primeiro formato de **EMBED** é a estrutura que deve ser encerrada com **ENDEMBED**. Essa estrutura pode conter qualquer código na *linguagem* permitida.

O segundo formato de **EMBED** é uma função autocontida. Os parâmetros de *code* contêm todo o código a ser executado, o que limita a utilidade a expressões bem simples.

Utilizando este form, você pode usar **EMBED** ao invés de **BEGINC++** para incorporar código C++ e especificar opções adicionais (por exemplo, **DISTRIBUTED**):

```
myFunction(string name) := EMBED(C++ [: options])
... text
ENDEMBED
```

ATENÇÃO: Esse recurso pode gerar corrupção de memória e/ou problemas de segurança. Portanto, recomendamos cautela e uma consideração detalhada. Consulte o Suporte Técnico antes de usar.

Exemplo:

```
//First form: a structure
IMPORT Python; //make Python language available
```

```
INTEGER addone(INTEGER p) := EMBED(Python)
# Python code that returns one more than the value passed to it
if p < 10:
    return p+1
else:
    return 0
ENDEMBED;

//Second form: a function
INTEGER addtwo(INTEGER p) := EMBED(Python, 'p+2');
```

Ver também: Estrutura BEGINC++, IMPORT, Função IMPORT

Estrutura FUNCTION

[resulttype] *funcname* (*parameterlist*) := **FUNCTION**

code

RETURN *retval*;

END;

<i>resulttype</i>	O tipo do valor de retorno da função. Se omitido, o tipo será obtido implicitamente da expressão <i>retval</i> .
<i>funcname</i>	O nome do atributo da ECL para a função.
<i>parameterlist</i>	Uma lista separada por vírgulas com os parâmetros a serem passados para a <i>função</i> . Estão disponíveis para todos os atributos definidos no <i>code</i> da FUNCTION.
<i>code</i>	As definições de atributos locais que compõem a função. Os atributos não podem ser EXPORT ou SHARED, mas podem incluir ações (como OUTPUT).
RETURN	Especifica a expressão do valor de retorno da função <i>retval</i> .
<i>retval</i>	O valor, a expressão, o conjunto de registros, a linha (registro) ou a ação a ser retornada.

A estrutura **FUNCTION** permite passar parâmetros para um conjunto de definições de atributos relacionados. Isso possibilita passar parâmetros para um atributo que é definido em termos de outros atributos não exportados, sem necessidade de parametrizar todos eles.

As ações de efeitos colaterais contidas no *code* da FUNCTION devem ter nomes de definição aos quais a função WHEN faz referência para executar.

Exemplo:

```
EXPORT doProjectChild(parentRecord l,UNSIGNED idAdjust2) := FUNCTION
  newChildRecord copyChild(childRecord l) := TRANSFORM
    SELF.person_id := l.person_id + idAdjust2;
    SELF := l;
  END;

  RETURN PROJECT(CHOOSEN(l.children, numChildren),copyChild(LEFT));
END;

//And called from
SELF.children := doProjectChild(l, 99);

/*****
EXPORT isAnyRateGE(String1 rate) := FUNCTION
  SetValidRates := ['0','1','2','3','4','5','6','7','8','9'];
  IsValidTradeRate := ValidDate(Trades.trd_drpt) AND
    Trades.trd_rate >= rate AND
    Trades.trd_rate IN SetValidRates;
  ValidPHR := Prev_rate(phr_grid_flag = TRUE,
    phr_rate IN SetValidRates,
    ValidDate(phr_date));
  IsPHRGridRate := EXISTS(ValidPHR(phr_rate >= rate,
    AgeOf(phr_date)<=24));
  IsMaxPHRRate := MAX(ValidPHR(AgeOf(phr_date) > 24),
    Prev_rate.phr_rate) >= rate;
  RETURN IsValidTradeRate OR IsPHRGridRate OR IsMaxPHRRate;
END;

/*****
//a FUNCTION with side-effect Action
```

```
namesTable := FUNCTION
  namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
  END;
  o := OUTPUT('namesTable used by user <x>');
  ds := DATASET([{'x','y',22}],namesRecord);
  RETURN WHEN(ds,0);
END;

z := namesTable : PERSIST('z');
//the PERSIST causes the side-effect action to execute only when the PERSIST is re-built

OUTPUT(z);

//*****
//a coordinated set of 3 examples

NameRec := RECORD
  STRING5 title;
  STRING20 fname;
  STRING20 mname;
  STRING20 lname;
  STRING5 name_suffix;
  STRING3 name_score;
END;

MyRecord := RECORD
  UNSIGNED id;
  STRING  uncleanedName;
  NameRec Name;
END;

ds := DATASET('RTTEST::RowFunctionData', MyRecord, THOR);

STRING73 CleanPerson73(STRING inputName) := FUNCTION
  suffix :=[ ' 0',' 1',' 2',' 3',' 4',' 5',' 6',' 7',' 8',' 9',
    ' J',' JR',' S',' SR'];
  InWords := Std.Str.CleanSpaces(inputName);
  HasSuffix := InWords[LENGTH(TRIM(InWords))-1 ..] IN suffix;
  WordCount := LENGTH(TRIM(InWords,LEFT,RIGHT)) -
    LENGTH(TRIM(InWords,ALL)) + 1;
  HasMiddle := WordCount = 5 OR (WordCount = 4 AND NOT HasSuffix) ;
  Sp1 := Std.Str.Find(InWords,' ',1);
  Sp2 := Std.Str.Find(InWords,' ',2);
  Sp3 := Std.Str.Find(InWords,' ',3);
  Sp4 := Std.Str.Find(InWords,' ',4);
  STRING5 title := InWords[1..Sp1-1];
  STRING20 fname := InWords[Sp1+1..Sp2-1];
  STRING20 mname := IF(HasMiddle,InWords[Sp2+1..Sp3-1],'');
  STRING20 lname := MAP(HasMiddle AND NOT HasSuffix => InWords[Sp3+1..],
    HasMiddle AND HasSuffix => InWords[Sp3+1..Sp4-1],
    NOT HasMiddle AND NOT HasSuffix => InWords[Sp2+1..],
    NOT HasMiddle AND HasSuffix => InWords[Sp2+1..Sp3-1],
    '');
  STRING5 name_suffix := IF(HasSuffix,InWords[LENGTH(TRIM(InWords))-1..],'');
  STRING3 name_score := '';
  RETURN title + fname + mname + lname + name_suffix + name_score;
END;

//Example 1 - a transform to create a row from an uncleaned name
NameRec createRow(string inputName) := TRANSFORM
  cleanedText := LocalAddrCleanLib.CleanPerson73(inputName);
  SELF.title := cleanedText[1..5];
  SELF.fname := cleanedText[6..25];
  SELF.mname := cleanedText[26..45];
  SELF.lname := cleanedText[46..65];
  SELF.name_suffix := cleanedText[66..70];
```



```
    SELF.name_score := cleanedText[71..73];
END;

myRecord t(myRecord l) := TRANSFORM
    SELF.Name := ROW(createRow(l.uncleanedName));
    SELF := l;
END;

y := PROJECT(ds, t(LEFT));
OUTPUT(y);

//Example 2 - an attribute using that transform to generate the row.
NameRec cleanedName(String inputName) := ROW(createRow(inputName));
myRecord t2(myRecord l) := TRANSFORM
    SELF.Name := cleanedName(l.uncleanedName);
    SELF := l;
END;

y2 := PROJECT(ds, t2(LEFT));
OUTPUT(y2);

//Example 3 = Encapsulate the transform inside the attribute by
// defining a FUNCTION.
NameRec cleanedName2(String inputName) := FUNCTION

    NameRec createRow := TRANSFORM
        cleanedText := LocalAddrCleanLib.CleanPerson73(inputName);
        SELF.title := cleanedText[1..5];
        SELF.fname := cleanedText[6..25];
        SELF.mname := cleanedText[26..45];
        SELF.lname := cleanedText[46..65];
        SELF.name_suffix := cleanedText[66..70];
        SELF.name_score := cleanedText[71..73];
    END;

    RETURN ROW(createRow);
END;

myRecord t3(myRecord l) := TRANSFORM
    SELF.Name := cleanedName2(l.uncleanedName);
    SELF := l;
END;

y3 := PROJECT(ds, t3(LEFT));
OUTPUT(y3);

//Example using MODULE structure to return multiple values from a FUNCTION
OperateOnNumbers(Number1, Number2) := FUNCTION
    result := MODULE
        EXPORT Multiplied := Number1 * Number2;
        EXPORT Differenced := Number1 - Number2;
        EXPORT Summed := Number1 + Number2;
    END;
    RETURN result;
END;

OperateOnNumbers(23,22).Multiplied;
OperateOnNumbers(23,22).Differenced;
OperateOnNumbers(23,22).Summed;
```

Ver também: Estrutura MODULE, Estrutura TRANSFORM, WHEN

Estrutura FUNCTIONMACRO

[resulttype] *funcname* (*parameterlist*) := **FUNCTIONMACRO**

code

RETURN *retval*;

ENDMACRO;

<i>resulttype</i>	O tipo do valor de retorno da função. Se omitido, o tipo será obtido implicitamente da expressão <i>retval</i> .
<i>funcname</i>	O nome da definição da ECL para a função/macro.
<i>parameterlist</i>	Uma lista separada por vírgulas dos nomes (tokens) dos parâmetros a serem passados para a função/macro. Esses nomes são usados em <i>code</i> e <i>retval</i> para indicar onde os valores dos parâmetros passados serão substituídos quando a função/macro for usada. Os tipos de valores para esses parâmetros não são permitidos, mas é possível especificar valores padrão como constantes de string.
<i>code</i>	As definições locais que compõem a função. As definições não podem ser EXPORT ou SHARED, mas podem incluir ações (como OUTPUT).
RETURN	Especifica a expressão do valor de retorno da função <i>retval</i> .
<i>retval</i>	O valor, a expressão, o conjunto de registros, a linha (registro) ou a ação a ser retornada.

A estrutura **FUNCTIONMACRO** é uma ferramenta de geração de código, como a estrutura **MACRO**, combinada com os benefícios do encapsulamento de código da estrutura **FUNCTION**. Uma vantagem de **FUNCTIONMACRO** sobre a estrutura **MACRO** é que pode ser chamada no contexto de uma expressão, da mesma forma que **FUNCTION**.

Ao contrário da estrutura **MACRO**, não é necessário usar **#UNIQUENAME** para evitar conflitos de nomes de definição internos quando **FUNCTIONMACRO** é usada várias vezes dentro do mesmo escopo de visibilidade. No entanto, a palavra-chave **LOCAL** deve ser usada explicitamente dentro de **FUNCTIONMACRO** se um nome de definição em seu *code* pode também ter sido definido fora de **FUNCTIONMACRO** e dentro do mesmo escopo de visibilidade. **LOCAL** identifica claramente que a definição é limitada ao *code* dentro de **FUNCTIONMACRO**.

Exemplo:

Este exemplo demonstra **FUNCTIONMACRO** usada em um contexto de expressão. O exemplo também mostra como **FUNCTIONMACRO** pode ser chamada várias vezes de suas definições internas sem conflitos de nome:

```
EXPORT Field_Population(infile,infield,compareval) := FUNCTIONMACRO
  c1 := COUNT(infile,infield=compareval);
  c2 := COUNT(infile);
  RETURN DATASET([{'Total Records',c2},
                  {'Recs=' + #TEXT(compareval),c1},
                  {'Population Pct',(INTEGER)((c2-c1)/c2)* 100.0}],
                  {STRING15 valuetype,INTEGER val});
ENDMACRO;

ds1 := dataset([{'M'},{'M'},{'M'},{''},{''},{'M'},{''},{'M'},{'M'},{''}],{STRING1 Gender});
ds2 := dataset([{''},{'M'},{'M'},{''},{''},{'M'},{''},{''},{'M'},{''}],{STRING1 Gender});

OUTPUT(Field_Population(ds1,Gender,''));
OUTPUT(Field_Population(ds2,Gender,''));
```

Este exemplo demonstra o uso da palavra-chave **LOCAL** para evitar conflitos de nome com definições externas dentro do mesmo escopo de visibilidade de **FUNCTIONMACRO**:

Referência a Linguagem ECL

Estruturas Especiais

```
numPlus := 'this creates a syntax error without LOCAL in the FUNCTIONMACRO';
AddOne(num) := FUNCTIONMACRO
    LOCAL numPlus := num + 1;    //LOCAL required here
    RETURN numPlus;
ENDMACRO;

AddTwo(num) := FUNCTIONMACRO
    LOCAL numPlus := num + 2;    //LOCAL required here
    RETURN numPlus;
ENDMACRO;

numPlus;
AddOne(5);
AddTwo(8);
```

Ver também: Estrutura FUNCTION, Estrutura MACRO

Estrutura INTERFACE

interfacename [(*parameters*)] := **INTERFACE** [(*inherit*)]

members;

END;

<i>interfacename</i>	O nome de definição ECL da interface.
<i>parameters</i>	Opcional. Os parâmetros de entrada da interface.
<i>inherit</i>	Opcional. Uma lista separada por vírgula das estruturas da INTERFACE cujos <i>membros</i> devem ser herdados. Esse pode não ser um parâmetro especificado. Várias <i>interfaces</i> herdadas podem conter definições de mesmo nome se forem do mesmo tipo e receberem os mesmos parâmetros; porém, se esses <i>membros herdados</i> possuem valores definidos, o conflito é solucionado pela substituição daquele <i>membro</i> na instância atual.
<i>members</i>	Definições, que podem ser EXPORTed ou SHARED. Estes podem ser semelhantes aos campos definidos em uma estrutura RECORD onde apenas o tipo e o nome são definidos – a expressão que define o valor não deve ser incluída (exceto em alguns casos onde a própria expressão define o tipo da definição, como as estruturas TRANSFORM). Caso nenhum valor padrão seja definido para o <i>membro</i> , qualquer MODULE (MÓDULO) derivado da INTERFACE deve definir o valor para aquele <i>membro</i> antes que o MODULE (MÓDULO) possa ser usado. Isso não inclui outra INTERFACE ou estruturas MODULE (MÓDULO) abstratas.

A estrutura **INTERFACE** define um bloco estruturado de *membros* relacionados que pode ter sido especificado como um parâmetro único para consultas complexas – em vez de especificar cada atributo individualmente. Isso é semelhante a uma estrutura MODULE (MÓDULO) com a opção VIRTUAL, exceto que erros são fornecidos para definições de *membros* privados (não SHARED ou EXPORTed).

Uma INTERFACE é uma estrutura abstrata; uma instância concreta que deve ser definida antes de poder ser usada em uma consulta. Uma estrutura MODULE (MÓDULO) que herda a INTERFACE e define os valores para os *membros* cria a instância concreta para ser usada pela consulta.

Exemplo:

```
HeaderRec := RECORD
  UNSIGNED4 RecID;
  STRING20  company;
  STRING25  address;
  STRING25  city;
  STRING2   state;
  STRING5   zip;
END;

HeaderFile := DATASET([ {1,'ABC Co','123 Main','Boca Raton','FL','33487'},
                        {2,'XYZ Co','456 High','Jackson','MI','49202'},
                        {3,'ABC Co','619 Eaton','Jackson','MI','49202'},
                        {4,'XYZ Co','999 Yamato','Boca Raton','FL','33487'},
                        {5,'Joes Eats','666 Slippery Lane','Nether','SC','12345'}
                      ],HeaderRec);

//define an interface
IHeaderFileSearch := INTERFACE
  EXPORT STRING20  company_val;
  EXPORT STRING2   state_val;
  EXPORT STRING25  city_val := '';
END;
```

```
//define a function that uses that interface
FetchAddress(IHeaderFileSearch opts) := FUNCTION

    //define passed values tests
    CompanyPassed := opts.company_val <> '';
    StatePassed := opts.state_val <> '';
    CityPassed := opts.city_val <> '';

    //define passed value filters
    NFilter := HeaderFile.Company = opts.company_val;
    SFilter := HeaderFile.State = opts.state_val;
    CFilter := HeaderFile.City = opts.city_val;

    //define the actual filter to use based on the passed values
    filter := MAP(CompanyPassed AND StatePassed AND CityPassed
        => NFilter AND SFilter AND CFilter,
        CompanyPassed AND StatePassed
        => NFilter AND SFilter ,
        CompanyPassed AND CityPassed
        => NFilter AND CFilter,
        StatePassed AND CityPassed
        => SFilter AND CFilter,
        CompanyPassed => NFilter ,
        StatePassed => SFilter ,
        CityPassed => CFilter,
        TRUE);
    RETURN HeaderFile(filter);
END;

//*****
//then you can use the interface

InRec := {HeaderRec AND NOT [RecID,Address,Zip]};

//this MODULE creates a concrete instance
BatchHeaderSearch(InRec l) := MODULE(IHeaderFileSearch)
    EXPORT STRING20 company_val := l.company;
    EXPORT STRING2 state_val := l.state;
    EXPORT STRING25 city_val := l.city;
END;

//that can be used like this
FetchAddress(BatchHeaderSearch(ROW({'ABC Co',' ',''},InRec)));

//or we can define an input dataset
InFile := DATASET([{'ABC Co','Boca Raton','FL'},
    {'XYZ Co','Jackson','MI'},
    {'ABC Co',' ',''},
    {'XYZ Co',' ',''},
    {'Joes Eats',' ',''}
],InRec);

//and an output nested child structure
HeaderRecs := RECORD
    UNSIGNED4 Pass;
    DATASET(HeaderRec) Headers;
END;

//and allow PROJECT to run the query once for each record in InFile
HeaderRecs XF(InRec L, INTEGER C) := TRANSFORM
    SELF.Pass := C;
    SELF.Headers := FetchAddress(BatchHeaderSearch(L));
END;
batchHeaderLookup := PROJECT(InFile,XF(LEFT,COUNTER));
```

`batchHeaderLookup;`

Ver também: Estrutura MODULE, LIBRARY

Estrutura MACRO

[resulttype] *macroname* (*parameterlist*) := **MACRO**

tokenstream;

ENDMACRO;

<i>resulttype</i>	Opcional. O tipo de resultado da macro. O único tipo válido é DATASET. Se omitido e <i>tokenstream</i> não tiver definições de Atributo, a macro será tratada como retornando um valor (normalmente, INTEGER ou STRING).
<i>macroname</i>	O nome da função definida pela estrutura MACRO.
<i>parameterlist</i>	Uma lista separada por vírgulas dos nomes (tokens) dos parâmetros a serem passados para a macro. Esses nomes são usados em <i>tokenstream</i> para indicar onde os valores dos parâmetros passados serão substituídos quando a macro for usada. Os tipos de valores para esses parâmetros não são permitidos, mas é possível especificar valores padrão como constantes de string.
<i>tokenstream</i>	As definições de atributo ou as ações que a macro executará.

A estrutura **MACRO** possibilita criar uma função sem conhecer os tipos de valor dos parâmetros que em algum momento serão passados a ela. O uso mais comum é a execução de funções com datasets arbitrários.

Uma macro se comporta como se você digitasse o *tokenstream* na posição exata em que a macro é usada, por meio de substituição léxica. Os tokens definidos em *parameterlist* são substituídos pelo texto passado para a macro em todos os lugares em que ocorrem no *tokenstream*. Isso possibilita escrever uma definição de MACRO válida que pode ser chamada com um conjunto de parâmetros que resulta em erros obscuros no tempo de compilação.

Há dois tipos básicos de macro: Valor ou Atributo. Uma macro VALUE não contém nenhuma definição de Atributo. Portanto, pode ser usada em qualquer lugar em que o tipo de valor gerado é adequado ao uso. Uma macro de atributo contém definições de atributo (detectadas pela presença de := no *tokenstream*). Portanto, pode ser usada apenas onde uma definição de Atributo é válida (uma linha por si só) e um item de *parameterlist* deve normalmente nomear o Atributo a ser usado para conter o resultado da macro (para que qualquer código depois da chamada da macro possa usar o resultado).

Exemplo:

```
// This is a DATASET Value macro that results in a crosstab
DATASET CrossTab(File,X,Y) := MACRO
    TABLE(File,{X, Y, COUNT(GROUP)},X,Y)
ENDMACRO;
// and would be used something like this:
OUTPUT(CrossTab(Person,person.per_st,Person.per_sex))
// this macro usage is the equivalent of:
//   OUTPUT(TABLE(Person,{person.per_st,Person.per_sex,COUNT(GROUP)},
// person.per_st,Person.per_sex)
//The advantage of using this macro is that it can be re-used to
// produce another cross-tab without recoding
// The following macro takes a LeftFile and looks up a field of it in
// the RightFile and then sets a field in the LeftFile indicating if
// the lookup worked.
IsThere(OutFile ,RecType,LeftFile,RightFile,LinkId ,SetField ) := MACRO
    RecType Trans(RecType L, RecType R) := TRANSFORM
        SELF.SetField := IF(NOT R.LinkId,0,1);
        SELF := L;
    END;
    OutFile := JOIN(LeftFile,
```

```
        RightFile,
        LEFT.LinkId=RIGHT.LinkId,
        Trans(LEFT,RIGHT),LEFT OUTER);
ENDMACRO;

// and would be used something like this:
MyRec := RECORD
    Person.per_cid;
    Person.per_st;
    Person.per_sex;
    Flag:=FALSE;
END;
MyTable1 := TABLE(Person(per_first_name[1]='R'),MyRec);
MyTable2 := TABLE(Person(per_first_name[1]='R',per_sex='F'),MyRec);

IsThere(MyOutTable,MyRec,MyTable1,MyTable2,per_cid,Flag)

    // This macro call generates the following code:
    // MyRec Trans(MyRec L, MyRec R) := TRANSFORM
    // SELF.Flag := IF(NOT R.per_cid ,0,1);
    // SELF := L;
    // END;
    // MyOutTable := JOIN(MyTable1,
    // MyTable2,
    // LEFT.per_cid=RIGHT.per_cid,
    // Trans(LEFT,RIGHT),
    // LEFT OUTER );

OUTPUT(MyOutTable);
//*****
//This macro has defaults for its second and third parameters
MyMac(FirstParm,yParm='22',zParm='42') := MACRO
    FirstParm := yParm + zParm;
ENDMACRO;

// and would be used something like this:
    MyMac(Fred)
    // This macro call generates the following code:
    // Fred := 22 + 42;
    //*****
    //This macro uses #EXPAND

MAC_join(attrname, leftDS, rightDS, linkflags) := MACRO
    attrname := JOIN(leftDS,rightDS,#EXPAND(linkflags));
ENDMACRO;
MAC_join(J1,People,Property,'LEFT.ID=RIGHT.PeopleID,LEFT OUTER')
//expands out to:
// J1 := JOIN(People,Property,LEFT.ID=RIGHT.PeopleID,LEFT OUTER);
```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, #UNIQUENAME, #EXPAND

Estrutura MODULE

modulename [(*parameters*)] := **MODULE** [(*inherit*)] [, **VIRTUAL**] [, **LIBRARY**(*interface*)] [, **FORWARD**]

members;

END;

<i>modulename</i>	O nome de definição do ECL para o módulo.
<i>parameters</i>	Opcional. Os parâmetros a serem disponibilizados a todas as <i>definições</i> .
<i>inherit</i>	Uma lista delimitada por vírgula da INTERFACE ou das estruturas MODULE abstratas nas quais essa instância será baseada. A instância atual herda todos os <i>members</i> das estruturas de base. Esse pode não ser um parâmetro especificado.
<i>members</i>	As definições que compõem o módulo. Essas definições podem receber parâmetros, incluir ações (tais como OUTPUT), e podem usar os tipos de escopo EXPORT ou SHARED. Podem não incluir INTERFACE ou MODULEs abstratos (veja abaixo). Se a opção LIBRARY for especificada, as <i>definições</i> devem implementar exatamente os membros que foram EXPORTADOS EXPORTed da <i>interface</i> .
VIRTUAL	Opcional. Especifica que o MODULE define uma interface abstrata cujas <i>definições</i> não exigem que os valores sejam definidos para elas.
LIBRARY	Opcional. Especifica que o MODULE implementa uma definição de <i>interface</i> da biblioteca de consulta.
<i>interface</i>	Refere-se à INTERFACE que define os <i>parâmetros</i> especificados para a biblioteca de consulta.. Os <i>parâmetros</i> especificados para o MODULE devem corresponder exatamente aos parâmetros especificados para a <i>interface</i> determinada.
FORWARD	Opcional. Adia o processamento das definições até que elas sejam usadas. A adição de , FORWARD em um MODULE adia o processamento das definições neste módulo até que elas sejam usadas. Isso gera dois efeitos principais: Impede a obtenção de dependências para definições que nunca são usadas e permite que as definições anteriores se refiram às definições posteriores. Note: Referências circulares ainda são ilegais.

A estrutura **MODULE** é um contêiner que permite agrupar definições relacionadas. Os *parâmetros* especificados ao MODULE são compartilhados por todas as definições relacionadas dos *membros*. Isso é semelhante à estrutura FUNCTION, exceto pelo fato de que não há um RETURN.

As regras de visibilidade das definições

As regras de escopo para os *membros* são iguais as que foram descritas anteriormente na discussão **Visibilidade das definições**:

- As definições locais são visíveis apenas através da próxima definição EXPORT ou SHARED (incluindo *membros* da estrutura MODULE aninhada, se a próxima definição EXPORT ou SHARED for um MODULE).
- As definições SHARED são visíveis para todas as definições subsequentes dentro da estrutura (incluindo *membros* de quaisquer estruturas MODULE aninhadas), mas não fora dela.
- As definições EXPORT são visíveis dentro da estrutura MODULE (incluindo *membros* de quaisquer estruturas MODULE subsequentes aninhadas) e fora dela.

Quaisquer *membros* das definições EXPORT podem ser referenciados usando um nível adicional da sintaxe padrão object.property. Por exemplo, supondo que a estrutura EXPORT MyModuleStructure MODULE esteja contida em

um Módulo de repositório do ECL denominado MyModule, e que ele contém um *membro* de EXPORT denominado MyDefinition, você referenciaria essa *definição* como:

```
MyModule.MyModuleStructure.MyDefinition
```

```
MyMod := MODULE
  SHARED x := 88;
  y := 42;
  EXPORT InMod := MODULE //nested MODULE
    EXPORT Val1 := x + 10;
    EXPORT Val2 := y + 10;
  END;
END;

MyMod.InMod.Val1;
MyMod.InMod.Val2;
```

Ações de efeitos-colaterais no MODULE

As ações de efeitos colaterais são permitidas no MODULE apenas com o uso da função WHEN, como neste exemplo:

```
//An Example with a side-effect action
EXPORT customerNames := MODULE
  EXPORT Layout := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
  END;
  Act := OUTPUT('customer file used by user <x>');
  EXPORT File := WHEN(DATASET([{'x','y',22}],Layout),Act);
END;
BOOLEAN doIt := TRUE : STORED('doIt');
IF (doIt, OUTPUT(customerNames.File));
//This code produces two results: the dataset, and the string
```

Módulos concretos vs. abstratos (VIRTUAL)

UM MÓDULO pode conter uma mistura de *membros* VIRTUAL e não-VIRTUAL. As regras são:

- ALL TODOS os *membros* são VIRTUAIS se o MODULE tiver a opção VIRTUAL ou se for uma INTERFACE
- Um *membro* é VIRTUAL se for declarado com o uso das palavras-chave EXPORT VIRTUAL ou SHARED VIRTUAL
- Um *membro* é VIRTUAL se a definição do mesmo nome no módulo *herdado* for VIRTUAL.
- Alguns *membros* nunca podem ser virtuais – estruturas RECORD.

Todos os *membros* EXPORTed e SHARED de um módulo abstrato *herdado* podem ser substituído por um item de redefinição na instância atual, seja essa instância atual abstrata ou concreta. As definições substituídas devem corresponder exatamente ao tipo e parâmetros dos *membros* *herdados*. Várias interfaces *herdadas* podem conter definições de mesmo nome se forem do mesmo tipo e receberem os mesmos parâmetros; porém, se esses *membros* *herdados* possuem valores definidos, o conflito é solucionado pela substituição daquele *membro* na instância atual.

Módulos LIBRARY

Um MODULE com a opção LIBRARY define um conjunto relacionado de funções que devem ser usadas como uma biblioteca de consulta (veja as discussões sobre função LIBRARY e ação BUILD). Há várias restrições sobre o que pode ser incluído em uma biblioteca de consulta. São elas:

- Não deve conter ações de efeitos colaterais (como OUTPUT ou BUILD)
- Não deve conter definições com serviços de tarefa anexado a elas (tais como PERSIST, STORED, SUCCESS, etc.)

Pode apenas EXPORTAR:

- Definições de dataset/recordset
- Definições de datarow (tais como a função ROW)
- Definições de valor único e booleano

E pode NÃO exportar:

- Ações (como OUTPUT ou BUILD)
- Funções TRANSFORM
- Estrutura MODULE
- Definições MACRO

Exemplo:

```
EXPORT filterDataset(String search, Boolean onlyOldies) := MODULE
  f := namesTable; //local to the "g" definition
  SHARED g := IF (onlyOldies, f(age >= 65), f);
    //SHARED = visible only within the structure
  EXPORT included := g(surname != search);
  EXPORT excluded := g(surname = search);
    //EXPORT = visible outside the structure
END;

filtered := filterDataset('Halliday', TRUE);
OUTPUT(filtered.included, ,NAMED('Included'));
OUTPUT(filtered.excluded, ,NAMED('Excluded'));

//same result, different coding style:
EXPORT filterDataset(Boolean onlyOldies) := MODULE
  f := namesTable;
  SHARED g := IF (onlyOldies, f(age >= 65), f);
  EXPORT included(String search) := g(surname <> search);
  EXPORT excluded(String search) := g(surname = search);
END;

filtered := filterDataset(TRUE);
OUTPUT(filtered.included('Halliday'), ,NAMED('Included'));
OUTPUT(filterDataset(true).excluded('Halliday'), ,NAMED('Excluded'));

//VIRTUAL examples
Mod1 := MODULE,VIRTUAL //a fully abstract module
  EXPORT val := 1;
  EXPORT func(INTEGER sc) := val * sc;
END;

Mod2 := MODULE(Mod1) //instance
  EXPORT val := 3; //a concrete member, overriding default value
    //while func remains abstract
END;

Mod3 := MODULE(Mod1) //a fully concrete instance
  EXPORT func(INTEGER sc) := val + sc; //overrides inherited func
END;
```

```
OUTPUT(Mod2.func(5)); //result is 15
OUTPUT(Mod3.func(5)); //result is 6

//FORWARD example
EXPORT MyModule := MODULE, FORWARD
  EXPORT INTEGER foo := bar;
  EXPORT INTEGER bar := 42;
END;

MyModule.foo;
```

Ver também: Estrutura FUNCTION, Visibilidade das definições, Estrutura INTERFACE, LIBRARY, BUILD

Estrutura TRANSFORM

resulttype *funcname* (*parameterlist*) := TRANSFORM [, SKIP(*condition*)]

[*locals*]

SELF.*outfield* := *transformation*;

END;

TRANSFORM(*resulttype*, *assignments*)

TRANSFORM(*datarow*)

<i>resulttype</i>	O nome de um Atributo da estrutura RECORD que especifica o formato dos resultados da função. Você pode usar TYPEOF aqui para especificar um dataset. Nenhuma relação implícita do dataset de entrada é herdada.
<i>funcname</i>	O nome da função definida pela estrutura TRANSFORM.
<i>parameterlist</i>	Uma lista separada por vírgulas dos tipos de valores e rótulos dos parâmetros a serem passados para a função TRANSFORM. Normalmente, são registros de dataset ou parâmetros de COUNTER, mas não estão limitados a isso.
SKIP	Opcional. Especifica a <i>condition</i> na qual a operação da função TRANSFORM é ignorada.
<i>condition</i>	Uma expressão lógica que define em que circunstâncias a operação de TRANSFORM não ocorre. Pode usar dados de <i>parameterlist</i> da mesma forma que a expressão <i>transformation</i> .
<i>locals</i>	Opcional. Definições de Atributos locais, úteis dentro da função TRANSFORM. Podem ser definidos para receber parâmetros e podem usar qualquer parâmetro passado para TRANSFORM.
SELF	Especifica o conjunto de registros dos resultados gerados por TRANSFORM.
<i>outfield</i>	O nome de um campo na estrutura <i>resulttype</i> .
<i>transformation</i>	Uma expressão que especifica como produzir o valor de <i>outfield</i> . Isso pode incluir outras operações da função TRANSFORM (transformações aninhadas).
<i>assignments</i>	Uma lista delimitada por ponto e vírgula de definições SELF . <i>outfield</i> := <i>transformation</i> .
<i>datarow</i>	Um único registro a transformar. Normalmente, a palavra-chave LEFT.

A estrutura **TRANSFORM** possibilita operações que devem ser executadas em datasets inteiros (como um JOIN) e qualquer tipo iterativo de processamento de registros (PROJECT, ITERATE, etc.). Uma estrutura TRANSFORM define as operações específicas que devem ocorrer registro a registro. Ela define a função chamada a cada vez que a operação que usa TRANSFORM precisa processar registro(s). Uma função TRANSFORM pode ser definida em termos de outra, e elas podem ser aninhadas.

A estrutura TRANSFORM especifica exatamente como cada campo do conjunto de resultados deve receber seu valor. O valor do resultado pode ser simplesmente o valor de um campo em um conjunto de registros de entrada ou pode ser o resultado de algum cálculo complexo ou avaliação de expressão condicional.

A estrutura TRANSFORM em si é uma ferramenta genérica. Cada operação que usa uma função TRANSFORM define o que TRANSFORM precisa receber e qual funcionalidade básica deve fornecer. Portanto, o caminho correto para compreender estruturas TRANSFORM é entender como são usadas pela função que chama a estrutura. Cada função que usa um TRANSFORM documenta o tipo necessário para alcançar o objetivo, embora a própria TRANSFORM também possa fornecer funcionalidades adicionais e receber outros parâmetros além dos necessários para a operação.

A opção SKIP especifica a *condition* que não gera resultados dessa iteração de TRANSFORM. No entanto, os valores COUNTER são incrementados mesmo quando SKIP elimina a criação do registro atual.

Definições dos Atributos de Transformação

As definições de atributos dentro da estrutura TRANSFORM são usadas para converter os dados passados como parâmetros para o formato *resulttype* nos resultados. Cada campo no layout do registro *resulttype* deve ser completamente definido no TRANSFORM. Você pode definir explicitamente cada campo usando a expressão *SELF.outfield* := *transformation*; , ou pode usar um destes atalhos:

```
SELF := [ ];
```

limpa campos nos resultados de *resulttype* que não foram definidos previamente na função de transformação, ao passo que este formato:

```
SELF.outfield := []; //the outfield names a child DATASET in  
// the resulttype RECORD Structure
```

limpa apenas os campos secundários em *outfield*, e este formato:

```
SELF := label; //the label names a RECORD structure parameter  
// in the parameterlist
```

define os resultados de cada campo no formato de resultados de *resulttype* que não foi definido como oriundo do campo *nomeado* correspondente do parâmetro label.

Você também pode definir atributos *local* dentro da estrutura TRANSFORM para organizar melhor o código. Esses atributos *local* podem receber parâmetros.

Funções TRANSFORM

Esse formato de TRANSFORM deve ser encerrado pela palavra-chave END. O *resulttype* deve ser especificado, e a própria função recebe parâmetros em *parameterlist*. Normalmente, esses parâmetros são estruturas RECORD, mas podem ser qualquer tipo de parâmetro, dependendo do tipo de função TRANSFORM esperado pela função que faz a chamada. O formato exato de uma função TRANSFORM está sempre associado diretamente à operação que usa essa função.

Exemplo:

```
Ages := RECORD  
  AgedRecs.id;
```

```
AgedRecs.id1;  
AgedRecs.id2;  
END;  
SequencedAges := RECORD  
  Ages;  
  INTEGER4 Sequence := 0;  
END;  
  
SequencedAges AddSequence(AgedRecs L, INTEGER C) :=  
  TRANSFORM, SKIP(C % 2 = 0) //skip even recs  
  INTEGER1 rangex(UNSIGNED4 divisor) := (L.id DIV divisor) % 100;  
  SELF.id1 := rangex(10000);  
  SELF.id2 := rangex(100);  
  SELF.Sequence := C;  
  SELF := L;  
END;  
  
SequencedAgedRecs := PROJECT(AgedRecs, AddSequence(LEFT,COUNTER));  
//Example of defining a TRANSFORM function in terms of another  
namesIdRecord assignId(namesRecord l, UNSIGNED value) := TRANSFORM  
  SELF.id := value;  
  SELF := l;  
END;  
  
assignId1(namesRecord l) := assignId(l, 1);  
  //creates an assignId1 TRANSFORM that uses assignId  
assignId2(namesRecord l) := assignId(l, 2);  
  //creates an assignId2 TRANSFORM that uses assignId
```

TRANSFORMs em linha

Este formato de TRANSFORM é usado em linha dentro da operação que usa a função. O *resulttype* deve ser especificado juntamente com todos os *assignments*. Esse formato é usado principalmente quando os *assignments* de transformação são triviais (como SELF := LEFT;).

Exemplo:

```
namesIdRecord assignId(namesRecord L) := TRANSFORM  
  SELF := L; //more like-named fields across  
  SELF := []; //clear all other fields  
END;  
  
projected1 := PROJECT(namesTable, assignId(LEFT));  
projected2 := PROJECT(namesTable, TRANSFORM(namesIdRecord,  
  SELF := LEFT;  
  SELF := []));  
//projected1 and projected2 do the same thing
```

TRANSFORMs abreviado em linha

Este formato de TRANSFORM é uma versão abreviada de TRANSFORMs em linha. Neste formato,

```
TRANSFORM(LEFT)
```

é diretamente equivalente a

```
TRANSFORM(RECORDOF(LEFT), SELF := LEFT)
```

Exemplo:

```
namesIdRecord assignId(namesRecord L) := TRANSFORM  
  SELF := L; //move like-named fields across
```

```
END;  
projected1 := PROJECT(namesTable, assignId(LEFT));  
projected2 := PROJECT(namesTable, TRANSFORM(namesIdRecord,  
      SELF := LEFT));  
projected3 := PROJECT(namesTable, TRANSFORM(LEFT));  
//projected1, projected2, and projected3 all do the same thing
```

Ver também: RECORD Structure, RECORDOF, TYPEOF, JOIN, PROJECT, ITERATE, ROLLUP, NORMALIZE, DENORMALIZE, FETCH, PARSE, ROW

Ações e Funções Built-in

ABS

ABS(*expression*)

<i>expression</i>	O valor (REAL ou INTEGER) para o qual o valor absoluto deve ser retornado.
Return:	ABS retorna um único valor do mesmo tipo que a expressão.

A função **ABS** retorna o valor absoluto da *expressão* (sempre um valor não negativo).

Exemplo:

```
AbsVal1 := ABS(1); // returns 1  
AbsVal2 := ABS(-1); // returns 1
```

ACOS

ACOS(*cosine*)

<i>cosine</i>	O valor coseno REAL para o qual o arco coseno deve ser localizado.
Return:	ACOS retorna um único valor REAL.

A função **ACOS** retorna o arco coseno (inverso) do *coseno*, em radianos.

Exemplo:

```
ArcCosine := ACOS(CosineAngle);
```

Ver também: COS, SIN, TAN, ASIN, ATAN, COSH, SINH, TANH

AGGREGATE

AGGREGATE(*recordset*, *resultrec*, *maintransform* [, *mergetransform* (**RIGHT1**, **RIGHT2**)] [, *groupingfields*] [, **LOCAL** | **FEW** | **MANY**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento.
<i>resultrec</i>	A estrutura RECORD do conjunto de registro do resultado.
<i>maintransform</i>	A função TRANSFORM a ser acionada para cada par de correspondência dos registros no <i>recordset</i> . Trata-se de uma operação local implícita em cada nó.
<i>mergetransform</i>	Opcional. A função TRANSFORM a ser acionada para realizar a fusão global dos registros do resultado no <i>maintransform</i> . Se omitida, o compilador tentará deduzir a fusão do <i>maintransform</i> .
<i>groupingfields</i>	Opcional. Uma lista de campos delimitada por vírgula no <i>recordset</i> a ser agrupado. Cada campo deve ser introduzido pela palavra-chave LEFT. Se omitida, todos os registros serão correspondentes.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior. Válido apenas se <i>mergetransform</i> for omitido.
FEW	Opcional. Indica que a expressão resultará em menos de 10.000 registros. Isso permite otimização para gerar um resultado significativamente mais rápido.
MANY	Opcional. Indica que a expressão resultará em mais de 10.000 registros.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	AGGREGATE retorna um conjunto de registros.

A função **AGGREGATE** é semelhante à ROLLUP, exceto quanto ao seu formato de resultado que não precisa corresponder ao formato de entrada. A função também é semelhante à TABLE, onde os *groupingfields* (se estiverem presente) determinam os registros correspondentes de forma que você obterá um resultado para cada valor único dos *groupingfields*. O *recordset* de entrada não precisa ter sido classificado pelos *groupingfields*.

A operação é implicitamente local, onde *maintransform* é acionado para processar os registros localmente em cada nó e os registros do resultado em cada nó são fundidos para gerar um resultado global.

Requerimentos da Função TRANSFORM - AGGREGATE

A *maintransform* deve adotar pelo menos um parâmetro: um registro LEFT de mesmo formato que a entrada *recset*. *recordset* registros RIGHT1 e RIGHT2 de mesmo formato que o *resultrec*. O formato do conjunto de registros resul-

tante deve ser o mesmo que o *resultrec*. LEFT se refere ao próximo registro de entrada e RIGHT ao resultado do transform anterior.

O *mergetransform* deve adotar pelo menos dois parâmetros: registros RIGHT1 e RIGHT2 de mesmo formato que o *resultrec*. O formato do conjunto de registros resultante deve ser o mesmo que o *resultrec*. RIGHT1 se refere ao resultado de *maintransform* em um nó e RIGHT2 ao resultado de *maintransform* em outro nó.

O *mergetransform* é gerado para expressões da forma:

```
SELF.x := <RIGHT.x <op> f(LEFT)
SELF.x := f(LEFT) <op> RIGHT.x
```

onde <op> é: MAX, MIN, SUM, +, &, |, ^, *

Como AGGREGATE funciona

No *maintransform*, LEFT se refere ao próximo registro de entrada e RIGHT ao resultado do transform anterior.

Há 4 casos interessantes:

(a) Se não houver correspondência de registros (e a operação não estiver agrupada), o resultado será de um único registro com todos os campos definidos para valores em branco.

(b) Se houver correspondência de um único registro, o primeiro registro correspondente aciona o *maintransform* da forma esperada.

(c) Se houver a correspondência de múltiplos registros em um único nó, os registros subsequentes correspondentes acionarão o *maintransform*, mas qualquer expressão no *maintransform* que não fizer referência ao registro RIGHT não será processada. Consequentemente, o valor desse campo é determinado pelo primeiro registro correspondente em vez do último.

(d) Se houver múltiplos registros correspondentes em múltiplos nós, a etapa (c) será desempenhada em cada nó e os registros de resumo serão fundidos. Isso exige um *mergetransform* que adota dois registros de tipo RIGHT. Sempre que possível, o gerador de código tenta deduzir o *mergetransform* do *maintransform*. Se ele não conseguir fazer isso, o usuário precisará especificar um.

```
inRecord := RECORD
  UNSIGNED box;
  STRING text{MAXLENGTH(10)};
END;
inTable := DATASET([ {1, 'Fred'}, {1, 'Freddy'},
                     {2, 'Freddi'}, {3, 'Fredrik'}, {1, 'FredJon'} ], inRecord);

//Example 1: Produce a list of box contents by concatenating a string:

outRecord1 := RECORD
  UNSIGNED box;
  STRING contents{MAXLENGTH(200)};
END;
outRecord1 t1(inRecord l, outRecord1 r) := TRANSFORM
  SELF.box := l.box;
  SELF.contents := r.contents + IF(r.contents <> '', ',', '') + l.text;
END;

outRecord1 t2(outRecord1 r1, outRecord1 r2) := TRANSFORM
  SELF.box := r1.box;
  SELF.contents := r1.contents + ',' + r2.contents;
END;
OUTPUT(AGGREGATE(inTable, outRecord1, t1(LEFT, RIGHT), t2(RIGHT1, RIGHT2), LEFT.box));
```

```
//This example could eliminate the merge transform if the SELF.contents expression in
//the t1 TRANSFORM were simpler, like this:
//    SELF.contents := r.contents + ',' + l.text;
//which would make the AGGREGATE function like this:
//    OUTPUT(AGGREGATE(inTable, outRecord1, t1(LEFT, RIGHT), LEFT.box));

//Example 2: A PIGMIX style grouping operation:
outRecord2 := RECORD
    UNSIGNED box;
    DATASET(inRecord) items;
END;
outRecord2 t3(inRecord l, outRecord2 r) := TRANSFORM
    SELF.box := l.box;
    SELF.items:= r.items + l;
END;
OUTPUT(AGGREGATE(inTable, outRecord2, t3(LEFT, RIGHT), LEFT.box));
```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, ROLLUP, TABLE

ALLNODES

ALLNODES(*operation*)

<i>operation</i>	O nome de um atributo ou código em linha que resulta em um DATASET ou INDEX.
Return:	ALLNODES retorna um conjunto de registros ou índice.

A função **ALLNODES** especifica que a *operação* é realizada em paralelo em todos os nós. **Essa função está disponível para uso apenas no Roxie.**

Exemplo:

```
ds := ALLNODES(JOIN(SomeData,LOCAL(SomeIndex), LEFT.ID = RIGHT.ID));
```

Ver também: THISNODE, LOCAL, NOLOCAL

APPLY

[*attrname* :=] **APPLY**(*dataset*, *actionlist* [, **BEFORE**(*actionlist*)] [, **AFTER**(*actionlist* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]))

<i>attrname</i>	Opcional. O nome da ação, que transforma a ação em definição de atributo, consequentemente não é executado até que <i>attrname</i> seja usado como uma ação.
<i>dataset</i>	O conjunto de registros para o qual a ação será aplicada. Pode ser o nome de um dataset físico ou um tipo compatível com essa operação.
<i>actionlist</i>	Uma lista delimitada por vírgula das ações a serem executadas simultaneamente. Normalmente, esse é um serviço externo (consulte a estrutura SERVICE). Este pode não ser um OUTPUT ou qualquer função que venha a acionar uma consulta secundária.
BEFORE	Especifica a execução do anexo <i>actionlist</i> antes que a primeira linha do dataset seja processada. Ainda não implementado no Thor; válido apenas no hthor e no Roxie.
AFTER	Especifica a execução da <i>actionlist</i> anexa após a última linha do dataset ter sido processada. Ainda não implementado no Thor; válido apenas no hthor e no Roxie.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

A ação **APPLY** desempenha todas as ações especificadas na *actionlist* em cada registro do *dataset* nominado. As ações são executadas na ordem em que aparecem na *actionlist*.

Exemplo:

```
EXPORT x := SERVICE
  echo(const string src):library='myfuncs',entrypoint='rtlEcho';
END;
APPLY(person,x.echo(last_name + ':' + first_name));
// concatenate each person's lastname and firstname and echo it
```

See Also: Estrutura SERVICE, DATASET

ASCII

ASCII(*recordset* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	ASCII retorna um recordset.

A função **ASCII** retorna o *recordset* com todos os campos da **STRING** traduzidos do **EBCDIC** para **ASCII**.

Exemplo:

```
AsciiRecs := ASCII(SomeEBCDICInput);
```

Ver também: **EBCDIC**

ASIN

ASIN(*sine*)

<i>sine</i>	O valor seno REAL para o qual o arco seno deve ser localizado.
Return:	ASIN retorna um valor REAL único.

A função **ASIN** retorna o arco seno (inverso) do *seno*, em radianos.

Exemplo:

```
ArcSine := ASIN(SineAngle);
```

Ver também: ACOS, COS, SIN, TAN, ATAN, COSH, SINH, TANH

ASSERT

ASSERT(*condition* [, *message*] [, **FAIL**] [, **CONST**])

ASSERT(*reset*, *condition* [, *message*] [, **FAIL**] [, **CONST**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

ASSERT(*reset*, *assertlist* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>condition</i>	A expressão lógica que deve sempre ser “true” (verdadeira).
<i>mensagem</i>	Opcional. O erro a ser exibido na workunit. Se omitida, uma mensagem será gerada a partir da localização aproximada no código e da condição que está sendo verificada.
FAIL	Opcional. Especifica que uma exceção foi gerada, finalizando imediatamente a workunit.
CONST	Opcional. Especifica que a condição é avaliada durante a geração do código.
<i>reset</i>	O conjunto de registros para o qual a condição deve ser verificada em relação a cada registro.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
<i>assertlist</i>	Uma lista delimitada por vírgula de ações ASSERT s da primeira forma, usada para verificar várias condições em relação a cada registro no <i>reset</i> .

A ação **ASSERT** avalia a *condição*, e se essa condição for falsa, a ação publicará a *mensagem* na workunit. Se a opção **FAIL** estiver presente, a workunit será finalizada imediatamente.

O form um é a form scalar que avalia a *condição* uma vez. O form dois avalia a *condição* uma vez para cada registro no *reset*. O form três é uma variante do form dois que aninha diversos **ASSERT**s do form um para que cada condição seja verificada em relação a cada registro no *reset*.

Exemplo:

```
val1 := 1;
val2 := 1;
val3 := 2;
val4 := 2 : STORED('val4');
ASSERT(val1 = val2);
```

```
ASSERT(val1 = val2, 'Abc1');
ASSERT(val1 = val3);
ASSERT(val1 = val3, 'Abc2');
ASSERT(val1 = val4);
ASSERT(val1 = val4, 'Abc3');
ds := DATASET([1,2],[INTEGER val1]) : GLOBAL;
// global stops advanced constant folding (if ever done)
ds1 := ASSERT(ds, val1 = val2);
ds2 := ASSERT(ds1, val1 = val2, 'Abc4');
ds3 := ASSERT(ds2, val1 = val3);
ds4 := ASSERT(ds3, val1 = val3, 'Abc5');
ds5 := ASSERT(ds4, val1 = val4);
ds6 := ASSERT(ds5, val1 = val4, 'Abc6');
OUTPUT(ds6);
ds7 := ASSERT(ds(val1 != 99),
  ASSERT(val1 = val2),
  ASSERT(val1 = val2, 'Abc7'),
  ASSERT(val1 = val3),
  ASSERT(val1 = val3, 'Abc8'),
  ASSERT(val1 = val4),
  ASSERT(val1 = val4, 'Abc9'));
OUTPUT(ds7);
rec := RECORD
  INTEGER val1;
  STRING text;
END;
rec t(ds 1) := TRANSFORM
  ASSERT(1.val1 <= 3);
  SELF.text := CASE(1.val1,1=>'One',2=>'Two',3=>'Three','Zero');
  SELF := 1;
END;
OUTPUT(PROJECT(ds, t(LEFT)));
```

Ver também: FAIL, ERROR

ASSTRING

ASSTRING(*bitmap*)

<i>bitmap</i>	O valor a ser tratado como uma string.
Return:	ASSTRING retorna um valor único da string.

A **função** ASSTRING retorna o *bitmap* como uma string. Isso é equivalente ao TRANSFER (*bitmap*,STRING_{*n*}), onde *n* corresponde ao mesmo número de bytes que os dados no *bitmap*.

Exemplo:

```
INTEGER1 MyInt := 65; //MyInt is an integer whose value is 65
MyVal1 := ASSTRING(MyInt); //MyVal1 is "A" (ASCII 65)
// this is directly equivalent to:
// STRING1 MyVal1 := TRANSFER(MyInt,STRING1); INTEGER1 MyVal3 := (INTEGER)MyVal1;
//MyVal3 is 0 (zero) because "A" is not a numeric character
```

Ver também: TRANSFER, Conversão de tipo

ATAN

ATAN(*tangent*)

<i>tangent</i>	O valor REAL da tangente para o qual o arco tangente deve ser localizado.
Return:	ATAB retorna um valor REAL único.

A função **ATAN** retorna o arco tangente (inverso) da *tangente*, em radianos.

Exemplo:

```
ArcTangent := ATAN(TangentAngle);
```

Ver também: ATAN2, ACOS, COS, ASIN, SIN, TAN, COSH, SINH, TANH

ATAN2

ATAN2()

y	O valor do numerador REAL da tangente.
x	O valor do denominador REAL da tangente.
Return:	ATAN2 retorna um valor REAL único.

A função **ATAN2** retorna o arco tangente (inverso) da tangente calculada, em radianos. Isso é semelhante à função **ATAN**, porém mais preciso e lida com situações onde x ou y é igual a zero. x ou y o padrão é zero.

Exemplo:

`ArcTangent := ATAN2(TangentNumerator, TangentDenominator);`

Ver também: **ATAN**, **ACOS**, **COS**, **ASIN**, **SIN**, **TAN**, **COSH**, **SINH**, **TANH**

AVE

AVE(*recordset*, *value* [, *expression*] [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

AVE(*valuelist*)

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Também pode ser a palavra-chave GROUP para indicar a média dos valores do campo em um grupo.
<i>value</i>	A expressão da qual o valor médio será localizado.
<i>expression</i>	Opcional. Uma expressão lógica indicando quais registros devem ser incluídos na média. Válido apenas quando o parâmetro <i>recordset</i> for a palavra-chave GROUP para indicar a média dos elementos em um grupo.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
<i>valuelist</i>	Uma lista delimitada por vírgula das expressões das quais o valor médio será localizado. Também pode ser um SET de valores.

Return: AVE retorna um único valor.

A função **AVE** retorna o *valor* médio (média aritmética) do *recordset* especificado ou a *valuelist*. Está configurada para retornar o valor zero caso o *recordset* esteja vazio.

Exemplo:

```
AvgBal1 := AVE(Trades, Trades.trd_bal);
AvgVal2 := AVE(4,8,16,2,1); //returns 6.2
SetVals := [4,8,16,2,1];
AvgVal3 := AVE(SetVals);    //returns 6.2
```

Ver também: MIN, MAX

BUILD

[*attrname* :=] **BUILD**(*baserecset*, [*indexrec*], *indexfile* [, *options*]);

[*attrname* :=] **BUILD**(*baserecset*, *keys*, *payload*, *indexfile* [, *options*]);

[*attrname* :=] **BUILD**(*indexdef* [, *options*]);

[*attrname* :=] **BUILD**(*indexdef*, *dataset*, [, *options*]);

BUILD(*library*);

<i>attrname</i>	Opcional. O nome da ação, que transforma a ação em definição de atributo, consequentemente não é executado até que <i>attrname</i> seja usado como uma ação.
<i>baserecset</i>	O conjunto de registro de dados para qual o arquivo de índice será criado. Pode ser um conjunto de registros derivado dos dados de base com os campos principais e a posição do arquivo.
<i>indexrec</i>	Opcional. A estrutura RECORD dos campos no <i>indexfile</i> que contém informações-chave e de posição do arquivo a serem mencionadas no <i>baserecset</i> . Os nomes e tipos de campos devem corresponder aos campos de <i>baserecset</i> (os tipos de campos REAL e DECIMAL não são permitidos). Também pode conter campos adicionais que não estejam presentes no <i>baserecset</i> . Se omitido, todos os campos em <i>baserecset</i> serão usados. O último campo deve ter o nome de um campo UNSIGNED8 definido usando a função {VIRTUAL(fileposition)} na declaração DATASET do <i>baserecset</i> .
<i>keys</i>	A estrutura RECORD dos campos no <i>indexfile</i> que contém informações-chave e de posição do arquivo a serem mencionadas no <i>baserecset</i> . Os nomes e tipos de campos devem corresponder aos campos de <i>baserecset</i> (os tipos de campos REAL e DECIMAL não são permitidos). Também pode conter campos adicionais que não estejam presentes no <i>baserecset</i> . Se omitido, todos os campos em <i>baserecset</i> serão usados.
<i>payload</i>	A estrutura RECORD do <i>indexfile</i> que contém campos adicionais não usados como chaves. Se o nome do <i>baserecset</i> estiver na estrutura, especificará "todos os outros campos ainda não nomeados no parâmetro Keys". Pode conter campos que não estejam presentes no <i>baserecset</i> . Esses campos não ocupam espaço nos nós não folha do índice e não podem ser citados em uma cláusula de filtro KEYED()
<i>indexfile</i>	Uma constante da string que contém o nome do arquivo lógico do índice a ser criado. Consulte a seção Escopo e Nomes de arquivos lógicos para obter mais detalhes sobre nomes de arquivos lógicos.
<i>options</i>	Opcional. Uma ou mais das opções listadas abaixo.
<i>indexdef</i>	O nome do atributo INDEX a ser compilado.
<i>library</i>	O nome de um atributo MODULE com a opção LIBRARY .

As primeiras quatro formas da ação **BUILD** criam arquivos de índice. Os índices são compactados automaticamente, minimizando a sobrecarga associada ao uso do acesso ao registro indexado. A palavra-chave BUILDINDEX pode ser usada no lugar de BUILD nestas formas.

A quinta forma cria uma biblioteca de – uma workunit que implementa a *biblioteca* especificada. Isso é semelhante à criação da extensão .DLL na programação do Windows, ou da .SO no Linux.

Opções de BUILD de Index

As opções a seguir estão disponíveis em todas as três formas BUILD de INDEX (apenas):

[, **CLUSTER**(*target*)] [, **SORTED**] [, **DISTRIBUTE**(*key*) [, **MERGE**]][, **DATASET**(*basedataset*)] [, **OVERWRITE**] [, **UPDATE**][, **EXPIRE**([*days*])][, **FEW**] [, **FILEPOSITION**(*false*)] [, **LOCAL**] [, **NOROOT**] [, **DISTRIBUTED**][, **COMPRESSED**(*LZW* | *ROW* | *FIRST*)] [, **WIDTH**(*nodes*)] [, **DEDUP**][, **SKEW**(*limit*, *target*)] [, **THRESHOLD**(*size*)]] [, **MAXLENGTH**(*value*)]][, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)][, **SET** (*option*, *value*)]

CLUSTER	Especifica o uso da <i>indexfile</i> especifica a gravação do nome de arquivo para a lista especificada de clusters de destino. Se omitido, o <i>indexfile</i> será gravado no cluster em que a tarefa é executada. O número de partes do arquivo físico gravado em disco sempre é determinado pelo número de nós no cluster onde a workunit é executada, independentemente do número de nós nos clusters de destino – a menos que a opção WIDTH também tenha sido especificada.
<i>target</i>	Uma lista de constantes de string delimitada por vírgulas que contém os nomes dos clusters no qual o arquivo <i>indexfile</i> será gravado. Os nomes devem estar listados como aparecem na página de Atividade do ECL Watch, ou como são retornados pela função Std.System.Thorlib.Group(); opcionalmente, podem apresentar colchetes contendo uma lista delimitada por vírgula dos números dos nós (baseado em 1) e/ou dos intervalos (especificados com um traço, como p.ex., n-m) para indicar o conjunto específico de nós para gravar.
SORTED	Especifica que <i>baserecset</i> já foi classificado, significando que a classificação automática com base em todos os campos <i>indexrec</i> não é exigida antes da criação do índice.
DISTRIBUTE	Especifica a compilação do <i>indexfile</i> com base na distribuição da chave.
<i>key</i>	O nome de uma definição do atributo INDEX existente.
MERGE	Opcional. Especifica a fusão do índice resultante na chave especificada.
DATASET	Necessário apenas quando <i>baserecset</i> for o resultado de uma operação (como um JOIN) cujo resultado a torna ambígua em relação a qual conjunto de dados físico está sendo indexada (em outras palavras, use essa opção apenas quando receber um erro que não possa ser deduzido). A nomeação do <i>basedataset</i> garante que os links de registro apropriados estão sendo usados no índice.
<i>basedataset</i>	O nome do atributo DATASET a partir do qual <i>baserecset</i> é derivado.
OVERWRITE	Especifica a substituição do <i>indexfile</i> caso ele exista.
UPDATE	especifica que o arquivo deve ser regravado apenas se houver alteração nos dados de código ou de entrada.
EXPIRE	Opcional. Especifica que se trata de um arquivo temporário que pode ser removido automaticamente após um determinado número de dias, após a leitura ter sido feita.
FILEPOSITION	Opcional. Se o <i>indicador</i> for FALSE, impede que o campo “fileposition” implícito seja criado e não tratará um campo inteiro à direita de forma diferente do resto da carga útil.
<i>flag</i>	Opcional. TRUE ou FALSE, indicando se o campo “fileposition” implícito será ou não criado.
<i>days</i>	Opcional. O número de dias contados a partir da última leitura do arquivo em que o arquivo será automaticamente removido. Se omitido, o padrão é sete (7).
FEW	Especifica que o <i>osindexfile</i> foi criado como um só arquivo de parte única. Usado apenas para datasets pequenos (normalmente arquivos do tipo pesquisa, como códigos de estado de 2 caracteres). Essa opção está atualmente obsoleta em função do uso de WIDTH (1).

Referência a Linguagem ECL
Ações e Funções Built-in

<i>indexdef</i>	O nome de uma definição do atributo INDEX existente que fornece os parâmetros <i>baserecset</i> , <i>indexrec</i> , e <i>indexfile</i> para uso.
LOCAL	Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
NOROOT	Especifica que o índice não é classificado em nível global e que não há índice de raiz para indicar qual parte do índice conterá uma entrada específica. Isso pode ser útil nas consultas Roxie juntamente com o uso de ALLNODES.
DISTRIBUTED	Especifica ambas as opções LOCAL e NOROOT (congruentes com a opção DISTRIBUTED em uma declaração INDEX, a qual especifica que o índice foi criado com as opções LOCAL e NOROOT).
COMPRESSED	Especifica o tipo de compactação usado. Se omitido, o padrão usado é LZW, uma variante do algoritmo Lempel-Ziv-Welch. A especificação de ROW compacta as entradas do índice com base nas diferenças entre linhas contíguas (usadas apenas com registros de comprimento fixo), e é recomendada para uso em situações onde o tempo mais rápido de descompactação é mais importante do que a quantidade de compactação alcançada. FIRST compacta os elementos principais comuns da chave (recomendado apenas para uso na comparação de cronometragem de tempo).
WIDTH	Especifica a gravação do <i>indexfile</i> para um número diferente de partes de arquivos físicos que o número de nós no cluster em que a tarefa é executada. Se omitido, o padrão será o número de nós no cluster no qual a workunit é executada. Esta opção serve principalmente para criar índices, destinados a serem implementados em um Roxie menor, em um Thor maior (tornando as consultas Roxie mais eficientes).
<i>nodes</i>	O número de partes do arquivo físico a serem gravadas. Se estiver definido para (1), essa opção operará exatamente da mesma forma que a opção FEW acima.
DEDUP	Especifica que as entradas duplicadas são eliminadas do INDEX.
SKEW	Indica que você sabe que os dados não serão espalhados uniformemente entre os nós (serão distorcidos e você opta por substituir o padrão especificando seu próprio valor limite para permitir que a tarefa continue, apesar da distorção).
<i>limit</i>	Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima de distorção a ser permitida antes que a tarefa falhe (a distorção padrão é 1,0 / <número de escravos no cluster>).
<i>target</i>	Opcional. Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima de distorção desejada a ser permitida (a distorção padrão é 1,0 / <número de escravos no cluster>).
THRESHOLD	Indica o tamanho mínimo de uma única parte antes que o limite SKEW seja aplicado.
<i>size</i>	Um valor inteiro indicando o número mínimo de bytes para uma parte única. O padrão é 1.
MAXLENGTH	Opcional. Esta opção é usada para criar índices que são compatíveis com versões anteriores às versões 3.0. Especifica o comprimento máximo de um registro de índice de comprimento variável. Os registros de comprimento fixo sempre utilizam o tamanho mínimo exigido. Se o comprimento máximo padrão causar problemas de ineficiência, ele pode ser substituído de forma explícita.
<i>value</i>	Opcional. Um valor inteiro que indica o comprimento máximo. Se omitido, o tamanho máximo será calculado a partir da estrutura do registro. Registros de comprimento variável que não especificam MAXLENGTH podem ser ligeiramente ineficientes.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.

Referência a Linguagem ECL

Ações e Funções Built-in

ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando as <i>threads</i> numthreads.
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
SET	Opcional. SET é usado para definir um valor para uma opção de metadata nomeada. Isso permite que você defina metadados do usuário cujo uso e propósito estejam à altura do desenvolvedor. Atualmente <i>_nodeSize</i> atual é o único metadata definido pelo sistema, embora outros nomes que começam com sublinhado (_) devem ser considerados como reservados para uso do sistema. É possível usar SET('_nodeSize', '32768') se seu hardware e padrão de uso trabalharam melhor com tamanhos de página maiores. O padrão (8192) pode não ser ideal para todos os cenários em hardware mais atuais. Recomendamos usar uma potência de 2 e não menor que 8k.
<i>option</i>	Uma constante de string – que faz distinção entre maiúsculas e minúsculas – que contém o nome da opção a ser definida.
<i>value</i>	O valor para o qual a opção será definida. Pode ser qualquer tipo de valor, dependendo do tipo esperado pela opção.

BUILD um Index de Acesso

[*attrname* :=] **BUILD**(*baserecset*, [*indexrec*], *indexfile* [, *options*]);

A forma 1 cria um arquivo de índice que permite o acesso com chave ao *baserecset*. O índice é usado principalmente pelas operações FETCH e JOIN (com a opção KEYED option).

Exemplo:

```
Vehicles := DATASET('vehicles',
  {STRING2 st,
   STRING20 city,
   STRING20 lname,
   UNSIGNED8 filepos{VIRTUAL(fileposition)}},
  FLAT);
BUILD(Vehicles, {lname, filepos}, 'vkey::lname');
//build key into Vehicles dataset on last name
```

BUILD um Index Payload

[*attrname* :=] **BUILD**(*baserecset*, *keys*, *payload*, *indexfile* [, *options*]);

A forma 2 cria um arquivo de índice contendo campos de *payload* útil adicionais além das *chaves*. Essa forma é usada principalmente em operações JOIN com “Half-key” para eliminar a necessidade de acessar diretamente o *baserecset*. Assim, o desempenho é superior ao da versão da mesma operação com “chave completa” (executada com a opção KEYED no JOIN).

Por padrão, os campos de *payload* são classificados durante a operação BUILDINDEX para minimizar o espaço nos nós folha da chave. Essa classificação pode ser controlada usando *sortIndexPayload* em uma declaração #OPTION .

Exemplo:

```
Vehicles := DATASET('vehicles',
  {STRING2 st,
   STRING20 city,
   STRING20 lname,
   UNSIGNED8 filepos{VIRTUAL(fileposition)}}},
  FLAT);
BUILD(Vehicles, {st,city}, {lname}, 'vkey::st.city');
//build key into Vehicles dataset on state and city
//payload the last name
```

BUILD a partir de uma definição INDEX

[*attrname* :=] **BUILD**(*indexdef* [, *options*]);

A forma 3 cria um arquivo de índice usando uma definição INDEX previamente especificada.

Exemplo:

```
nameKey := INDEX(mainTable, {surname, forename, filepos}, 'name.idx');
BUILD(nameKey); //gets all info from the INDEX definition
```

[*attrname* :=] **BUILD**(*indexdef*, *dataset* [, *options*]);

A forma 4 cria um arquivo de índice em um dataset usando uma definição INDEX previamente especificada.

Isso é usado para criar um índice cuja definição do dataset é complexa. Isso permite que o índice seja separado de forma lógica do dataset a partir do qual é criado. Isso é útil especificamente quando a definição do dataset é bastante complicada (Mb de origem), já que quando o índice é subsequentemente usado em uma consulta, todos os códigos usados para criá-lo também são interpretados.

Exemplo:

```
ds = DATASET(100, TRANSFORM({ unsigned id }, SELF.id := COUNTER));
i := INDEX({ unsigned id }, 'myIndex');
BUILD(i, ds);
```

BUILD uma Biblioteca de Consulta

BUILD(*library*);

A forma 5 cria uma biblioteca de consulta externa **para ser usada apenas no Roxie**.

Uma biblioteca de consulta permite que um conjunto de atributos relacionados seja agrupado como uma unidade autocontida, para que o código possa ser compartilhado entre diferentes workunits. Isso reduz o tempo necessário para implementar um conjunto de atributos, podendo reduzir a pegada de memória para o grupo de consultas no Roxie que usam a *biblioteca*. Além disso, a funcionalidade na *biblioteca* pode ser atualizada sem a necessidade de reimplementar todas as consultas que utilizam essa funcionalidade.

As bibliotecas de consulta são adequadas para agrupar conjuntos de funções próximas. Não são adequadas para a inclusão de atributos definidos como MACROS – o significado de uma macro é desconhecido até que os seus parâmetros sejam substituídos.

A forma do nome da #WORKUNIT nomeia a workunit que BUILD cria como biblioteca externa. Este nome é o nome da biblioteca externa usada pela função LIBRARY (que fornece acesso à biblioteca a partir da consulta que está usando

a *biblioteca*). Uma vez que a própria *workunit* é a biblioteca de consulta externa, **BUILD** construir(*biblioteca*) deve ser a única ação na *workunit*.

Exemplo:

```
NamesRec := RECORD
  INTEGER1  NameID;
  STRING20  FName;
  STRING20  LName;
END;
FilterLibIfacel(DATASET(namesRec) ds, STRING search) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;

FilterDsLib1(DATASET(namesRec) ds, STRING search) :=
  MODULE, LIBRARY(FilterLibIfacel)
  EXPORT matches := ds(Lname = search);
  EXPORT others  := ds(Lname != search);
END;
#WORKUNIT('name', 'Ppass.FilterDsLib')
BUILD(FilterDsLib1);
```

Ver também: INDEX, JOIN, FETCH, MODULE, INTERFACE, LIBRARY, DISTRIBUTE, #WORKUNIT

CASE

CASE(*expression*, *caseval* => *value*, [... , *caseval* => *value*] [, *elsevalue*])

<i>expression</i>	Uma expressão que resulta em um único valor.
<i>caseval</i>	Um valor a ser comparado com o resultado da expressão.
=>	O operador "resulta em" - válido somente em CHOOSESETS, CASE e MAP.
<i>value</i>	O valor a ser retornado. Isso pode ser qualquer expressão ou ação.
<i>elsevalue</i>	Opcional. O valor a ser retornado quando o resultado da expressão não corresponde a nenhum dos valores <i>caseval</i> . Pode ser omitido se todos os valores de retorno forem ações (o padrão então seria nenhuma ação), ou se todos os valores de retorno forem conjuntos de registro (o padrão então seria um conjunto de registros vazio).
Return:	CASE retorna um único valor, um conjunto de valores, um conjunto de registros ou uma ação.

A função **CASE** avalia a *expressão* e retorna o *valor* cujo *caseval* corresponde ao resultado da *expressão* . Se não houver correspondência, ela retorna o *elsevalue*.

Deverá haver a quantidade de parâmetros de *caseval* => *value* necessária para especificar todos os valores esperados da *expressão* (deve haver pelo menos um). Todos os parâmetros do *valor* de retorno devem ser do mesmo tipo.

Exemplo:

```
MyExp := 1+2;
MyChoice := CASE(MyExp, 1 => 9, 2 => 8, 3 => 7, 4 => 6, 5);
// returns a value of 7 for the caseval of 3
MyRecSet := CASE(MyExp, 1 => Person(per_st = 'FL'),
  2 => Person(per_st = 'GA'),
  3 => Person(per_st = 'AL'),
  4 => Person(per_st = 'SC'),
  Person);
// returns set of Alabama Persons for the caseval of 3
MyAction := CASE(MyExp, 1 => FAIL('Failed for reason 1'),
  2 => FAIL('Failed for reason 2'),
  3 => FAIL('Failed for reason 3'),
  4 => FAIL('Failed for reason 4'),    FAIL('Failed for unknown reason'));
// for the caseval of 3, Fails for reason 3
```

Ver também: MAP, CHOOSE, IF, REJECTED, WHICH

CATCH

result := **CATCH**(*reset*, *action* [, **UNORDERED** | **ORDERED**(*bool*)][, **STABLE** | **UNSTABLE**][, **PARALLEL** [(*numthreads*)]][, **ALGORITHM**(*name*)]);

<i>result</i>	Nome de definição do conjunto de registro resultante
<i>reset</i>	A expressão do recordset que, se falhar, faz com que a <i>ação</i> seja iniciada.
<i>action</i>	Uma das três ações válidas abaixo.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	CATCH retorna um conjunto de registros (que pode estar vazio).

A função **CATCH** executa a *ação* se a expressão *reset* falhar por qualquer motivo.

As *ações* válidas são:

SKIP	Especifica ignorar o erro e dar continuidade, retornando um dataset vazio.
ONFAIL (<i>transform</i>)	Especifica o retorno de um único registro da função <i>transform</i> . A função TRANSFORM pode usar FAILCODE e/ou FAILMESSAGE para fornecer detalhes da falha e deve resultar em uma estrutura RECORD de mesmo formato que <i>reset</i> .
FAIL	A ação FAIL , que especifica a mensagem de erro a ser gerada. Isso serve para fornecer informações mais úteis ao usuário final sobre o motivo da falha da tarefa.

Exemplo:

```
MyRec := RECORD
    STRING50 Value1;
    UNSIGNED Value2;
END;

ds := DATASET([{'C',1},{ 'C',2},{ 'C',3},
               {'C',4},{ 'C',5},{ 'X',1},{ 'A',1}],MyRec);

MyRec FailTransform := transform
    self.value1 := FAILMESSAGE[1..17];
    self.value2 := FAILCODE
END;

limited1 := LIMIT(ds, 2);
limited2 := LIMIT(ds, 3);
limited3 := LIMIT(ds, 4);
```

```
recovered1 := CATCH(limited1, SKIP);  
recovered2 := CATCH(limited2, ONFAIL(FailTransform));  
recovered3 := CATCH(CATCH(limited3, FAIL(1, 'Failed, dude')), ONFAIL(FailTransform));  
  
OUTPUT(recovered1); //empty recordset  
OUTPUT(recovered2); //  
OUTPUT(recovered3); //
```

Ver também: Estrutura TRANSFORM, FAIL, FAILCODE, FAILMESSAGE

CHOOSE

CHOOSE(*expression*, *value*,... , *value*, *elsevalue*)

<i>expressão</i>	Uma expressão aritmética que resulta em um valor inteiro positivo e que determina qual parâmetro de valor será retornado.
<i>value</i>	Os valores a serem retornados. Deverá haver a quantidade de parâmetros de valor necessária para especificar todos os valores esperados da expressão. Isso pode ser qualquer expressão ou ação.
<i>elsevalue</i>	O valor a ser retornado quando a expressão retorna um valor fora do intervalo. O último parâmetro é sempre <i>elsevalue</i> .
Return:	CHOOSE retorna um único valor.

A função **CHOOSE** avalia a *expressão* e retorna o parâmetro de *valor* cuja posição ordinal na lista de parâmetros corresponde ao resultado da *expressão*. Se não houver correspondência, ela retorna o *elsevalue*. Todos os *valores* e *elsevalue* devem ser do mesmo tipo.

Exemplo:

```
MyExp := 1+2;
MyChoice := CHOOSE(MyExp,9,8,7,6,5); // returns 7
MyChoice := CHOOSE(MyExp,1,2,3,4,5); // returns 3
MyChoice := CHOOSE(MyExp,15,14,13,12,11); // returns 13
WorstRate := CHOOSE(IntRate,1,2,3,4,5,6,6,6,6,0);
// WorstRate receives 6 if the IntRate is 7, 8, or 9
```

Ver também: CASE, IF, MAP

CHOOSEN

CHOOSEN(*recordset*, *n* [, *startpos*] [, **FEW**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
<i>n</i>	O número de registros a ser retornado. Se for zero (0), nenhum registro será retornado, e se ALL ou CHOOSEN:ALL estiver presente, todos os registros serão retornados. A opção CHOOSEN:ALL é uma constante que pode ser usada em qualquer expressão.
<i>startpos</i>	Opcional. A posição ordinal no recordset do primeiro registro a ser retornado. Se omitido, o padrão é um (1).
FEW	Opcional. Especifica converter internamente para uma operação TOPN se <i>n</i> for um número variável (um atributo ou parâmetro especificado) e se o conjunto de registros de entrada vier a partir de um SORT.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	CHOOSEN retorna um conjunto de registros.

A função **CHOOSEN** (choose-*n*) retorna o primeiro número *n* de registros, começando com o registro no *startpos*, a partir do *recordset* especificado.

Exemplo:

```
AllRecs := CHOOSEN(Person,ALL); // returns all recs from Person
FirstFive := CHOOSEN(Person,5); // returns first 5 recs from Person
NextFive := CHOOSEN(Person,5,6); // returns next 5 recs from Person
LimitRecs := CHOOSEN(Person,IF(MyLimit<>0,MyLimit,CHOOSEN:ALL));
```

Ver também: SAMPLE, CHOOSESETS

CHOOSESETS

CHOOSESETS(*reset*, *condition* => *n* [, *o*] [, **EXCLUSIVE** | **LAST** | **ENTH** [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>reset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um record set derivado de algumas condições de filtro, ou qualquer expressão que resulte em um record set derivado.
<i>condition</i>	A expressão lógica que define quais registros serão incluídos no result set.
=>	O operador "resulta em" - válido somente em CHOOSESETS, CASE e MAP.
<i>n</i>	O número máximo de registros a ser retornado. Se o número for zero (0), não será retornado nenhum registro que atenda à condição.
<i>o</i>	Opcional. O número máximo de registros a ser retornado que não atende à nenhuma das condições especificadas.
EXCLUSIVE	Opcional. Especifica que os parâmetros da condição são mutualmente exclusivos.
LAST	Opcional. Especifica a escolha dos últimos <i>n</i> registros que atendem à condição em vez do primeiro <i>n</i> . Essa opção é implicitamente EXCLUSIVE .
ENTH	Opcional. Especifica a escolha de uma amostragem de registros que atendem à condição em vez do primeiro <i>n</i> . Essa opção é implicitamente EXCLUSIVE .
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for "False" (Falso), especifica que a ordem do registro de resultado não é importante. Quando for "True" (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	CHOOSESETS retorna um conjunto de registros.

A função **CHOOSESETS** retorna um conjunto de registros do *reset*. O conjunto de resultado está limitado ao número *n* de registros que atendem à *condição* listada. CHOOSESETS pode adotar quantos parâmetros de *condição* => *n* forem necessários para especificar exatamente o conjunto de registros desejado. Esta é uma forma abreviada de concatenar os conjuntos do resultado dos múltiplos acionamentos da função **CHOOSE** para o mesmo *reset* com *condições* de filtro distintas, porém CHOOSESETS executa significativamente mais rápido. Esta técnica também é conhecida como "cutback."

Exemplo:

```
MyResultSet := CHOOSESETS(Person,
    per_first_name = 'RICHARD' => 100,
    per_first_name = 'GWENDOLYN' => 200, 100)
// returns a set containing 100 Richards, 200 Gwendolyns, 100 others
```

Ver também: **CHOOSE**, **SAMPLE**

CLUSTER SIZE

CLUSTER SIZE

Return:	CLUSTER SIZE retorna um único valor INTEGER.
---------	--

A constante de tempo de compilação **CLUSTER SIZE** retorna o número de nós em um cluster. Este é o mesmo valor retornado pela função Std.System.ThorLib.Nodes(). Este é o mesmo valor retornado pela função Std.System.ThorLib.Nodes() .

Exemplo:

```
OUTPUT (CLUSTER SIZE)
```

COMBINE

COMBINE(*leftrecset*, *rightrecset* [, *transform*][,LOCAL])

COMBINE(*leftrecset*, *rightrecset*, **GROUP** , *transform* [,LOCAL] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>leftrecset</i>	O recordset LEFT.
<i>rightrecset</i>	O recordset RIGHT.
<i>transform</i>	O acionamento da função TRANSFORM. Se omitida, COMBINE retorna todos os campos de ambos <i>leftrecset</i> e <i>rightrecset</i> com o segundo de qualquer campo de nome em duplicidade removido.
LOCAL	A opção LOCAL é obrigatória quando COMBINE for usado no Thor (e estiver implícito no hThor/Roxie).
GROUP	Especifica que <i>rightrecset</i> foi GROUPed. Se não for este o caso, ocorrerá um erro.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	COMBINE retorna um conjunto de registros.

A função **COMBINE** combina *leftrecset* e *rightrecset* em uma base de registro por registro na ordem em que aparecem em cada um.

Requerimentos da Função COMBINE TRANSFORM

Para a forma 1 do COMBINE, a função transform deve adotar pelo menos dois parâmetros: um registro LEFT que deve estar no mesmo formato que o *leftrecset* e um RIGHT cujo formato deve ser o mesmo que *rightrecset*. O formato do conjunto de registros resultante deve ser diferente das entradas.

Para a forma 2, a função transform deve adotar pelo menos três parâmetros: um registro LEFT que deve estar no mesmo formato que o *leftrecset*, um um registro RIGHT que deve estar no mesmo formato que o *rightrecset* e um ROWS(RIGHT) cujo formato deve ser o de um parâmetro DATASET(RECORDOF(*rightrecset*)). O formato do conjunto de registros resultante deve ser diferente das entradas.

Forma 1 do COMBINE

A forma 1 do COMBINE gera seus resultados através da especificação de cada registro do *leftrecset* (juntamente com o registro na mesma posição ordinal dentro do *rightrecset*) para o *transform* a fim de gerar um único registro de

resultado. O agrupamento (se houver) no *leftrecset* é preservado. Ocorrerá um erro se *leftrecset* e *rightrecset* tiverem um número diferente de registros.² d

Exemplo:

```
inrec := RECORD
  UNSIGNED6 uid;
END;
outrec := RECORD(inrec)
  STRING20 name;
  STRING11 ssn;
  UNSIGNED8 dob;
END;
ds := DATASET([1,2,3,4,5,6], inrec);
i1 := DATASET([ {1, 'Kevin'},
  {2, 'Richard'},
  {5, 'Nigel'} ],
  {UNSIGNED6 udid, STRING10 name });
i2 := DATASET([ {3, '000-12-3462'},
  {5, '000-12-8723'},
  {6, '000-10-1002'} ],
  {UNSIGNED6 udid, STRING10 ssn });
i3 := DATASET([ {1, 19700117},
  {4, 19831212},
  {6, 20010101} ],
  {UNSIGNED6 udid, UNSIGNED8 dob});
j1 := JOIN(ds, i1, LEFT.udid = RIGHT.udid, LEFT OUTER, LOOKUP);
j2 := JOIN(ds, i2, LEFT.udid = RIGHT.udid, LEFT OUTER, LOOKUP);
j3 := JOIN(ds, i3, LEFT.udid = RIGHT.udid, LEFT OUTER, LOOKUP);
combined1 := COMBINE(j1, j2,
  TRANSFORM(outRec,
    SELF := LEFT;
    SELF := RIGHT;
    SELF := []));
LOCAL;
combined2 := COMBINE(combined1, j3,
  TRANSFORM(outRec,
    SELF.dob := RIGHT.dob;
    SELF := LEFT));
LOCAL;
OUTPUT(combined1);
OUTPUT(combined2);
```

Forma do COMBINE

A forma 2 do COMBINE gera seus resultados através da especificação de cada registro do *leftrecset* (o grupo na mesma posição ordinal dentro do *rightrecset* – juntamente com o primeiro registro no grupo) para o *transform* a fim de gerar um único registro de resultado. O agrupamento (se houver) no *leftrecset* é preservado. *leftrecset*. Ocorrerá um erro se o número de registros no *leftrecset* for distinto do número de grupos no *rightrecset*.

Exemplo:

```
inrec := {UNSIGNED6 udid};
outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;
nameRec := RECORD
  STRING20 name;
END;
resultRec := RECORD(inrec)
  DATASET(nameRec) names;
```

```
END;
ds := DATASET([1,2,3,4,5,6], inrec);
dsg := GROUP(ds, ROW);
i1 := DATASET([ {1, 'Kevin' ,10},
                {2, 'Richard', 5},
                {5, 'Nigel' , 2},
                {0, ' ', 0} ], outrec);
i2 := DATASET([ {1, 'Kevin Hall', 12},
                {2, 'Richard Chapman', 15},
                {3, 'Jake Smith', 20},
                {5, 'Nigel Hicks', 100},
                {0, ' ', 0} ], outrec);
i3 := DATASET([ {1, 'Halligan', 8},
                {2, 'Richard', 8},
                {6, 'Pete', 4},
                {6, 'Peter', 8},
                {6, 'Petie', 1},
                {0, ' ', 0} ], outrec);
j1 := JOIN(dsg,i1,
           LEFT.uid = RIGHT.uid,
           TRANSFORM(outrec,
                     SELF := LEFT;
                     SELF := RIGHT),
           LEFT OUTER, MANY LOOKUP);
j2 := JOIN(dsg,i2,
           LEFT.uid = RIGHT.uid,
           TRANSFORM(outrec,
                     SELF := LEFT;
                     SELF := RIGHT),
           LEFT OUTER, MANY LOOKUP);
j3 := JOIN(dsg,i3,
           LEFT.uid = RIGHT.uid,
           TRANSFORM(outrec,
                     SELF := LEFT;
                     SELF := RIGHT),
           LEFT OUTER, MANY LOOKUP);
combined := REGROUP(j1, j2, j3);
resultRec t(inrec l, DATASET(RECORDOF(combined)) r) := TRANSFORM
  SELF.names := PROJECT(r, TRANSFORM(nameRec, SELF := LEFT));
  SELF := l;
END;
res1 := COMBINE(dsg,combined,GROUP,t(LEFT, ROWS(RIGHT)(score != 0)),LOCAL);
OUPUT(res1);

//A variation using rows in a child query.
resultRec t2(inrec l, DATASET(RECORDOF(combined)) r) := TRANSFORM
  SELF.names := PROJECT(SORT(r, -score),
                        TRANSFORM(nameRec, SELF := LEFT));
  SELF := l;
END;
res2 := COMBINE(dsg,combined,GROUP,t2(LEFT,ROWS(RIGHT)(score != 0)),LOCAL);
OUPUT(res2);
```

Ver também: GROUP, REGROUP

CORRELATION

CORRELATION(*recset*, *valuex*, *valuey* [, *expression*] [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Também pode ser a palavra-chave GROUP para indicar a operação nos elementos em cada grupo, quando usada em uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>valuex</i>	Um campo ou expressão numérica.
<i>valuey</i>	Um campo ou expressão numérica.
<i>expression</i>	Opcional. Uma expressão lógica indicando quais registros devem ser incluídos no cálculo. Válido apenas quando o parâmetro <i>recset</i> for a palavra-chave GROUP .
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	CORRELATION retorna um valor REAL único.

A função **CORRELATION** retorna o Coeficiente de Correlação Produto-Momento de Pearson entre *valuex* e *valuey*.

Exemplo:

```
pointRec := { REAL x, REAL y };
analyse( ds ) := MACRO
#uniquename(stats)
%stats% := TABLE(ds, { c      := COUNT(GROUP),
    sx    := SUM(GROUP, x),
    sy    := SUM(GROUP, y),
    sxx   := SUM(GROUP, x * x),
    sxy   := SUM(GROUP, x * y),
    syy   := SUM(GROUP, y * y),
    varx  := VARIANCE(GROUP, x);
    vary  := VARIANCE(GROUP, y);
    varxy := COVARIANCE(GROUP, x, y);
    rc    := CORRELATION(GROUP, x, y) });
OUTPUT(%stats%);
// Following should be zero
OUTPUT(%stats%, { varx - (sxx-sx*sx/c)/c,
    vary - (syy-sy*sy/c)/c,
```



```
    varxy - (sxy-sx*sy/c)/c,  
    rc - (varxy/SQRT(varx*vary)) }));  
OUTPUT(%stats%, { 'bestFit: y=' +  
  (STRING)((sy-sx*varxy/varx)/c) +  
  ' + ' +  
  (STRING)(varxy/varx)+'x' }));  
ENDMACRO;  
ds1 := DATASET([ {1,1}, {2,2}, {3,3}, {4,4}, {5,5}, {6,6}], pointRec);  
ds2 := DATASET([ {1.93896e+009, 2.04482e+009},  
  {1.77971e+009, 8.54858e+008},  
  {2.96181e+009, 1.24848e+009},  
  {2.7744e+009, 1.26357e+009},  
  {1.14416e+009, 4.3429e+008},  
  {3.38728e+009, 1.30238e+009},  
  {3.19538e+009, 1.71177e+009} ], pointRec);  
ds3 := DATASET([ {1, 1.00039},  
  {2, 2.07702},  
  {3, 2.86158},  
  {4, 3.87114},  
  {5, 5.12417},  
  {6, 6.20283} ], pointRec);  
analyse(ds1);  
analyse(ds2);  
analyse(ds3);
```

Ver também: VARIANCE, COVARIANCE

COS

COS(*angle*)

<i>angle</i>	O valor radiano REAL para o qual o coseno deve ser localizado.
Return:	COS retorna um valor REAL único.

A função **COS** retorna o coseno do *ângulo*.

Exemplo:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
Cosine45 := COS(Angle45);   //get cosine of the 45 degree angle
```

Ver também: ACOS, SIN, TAN, ASIN, ATAN, COSH, SINH, TANH

COSH

COSH(*angle*)

<i>angle</i>	O valor radiano REAL para o qual o coseno hiperbólico deve ser localizado.
Return:	COSH retorna um valor REAL único.

A função **COSH** retorna o coseno hiperbólico do *ângulo*.

Exemplo:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
HyperbolicCosine45 := COSH(Angle45);
                        //get hyperbolic cosine of the 45 degree angle
```

Ver também: ACOS, SIN, TAN, ASIN, ATAN, COS, SINH, TANH

COUNT

COUNT(*recordset* [, *expression*] [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

COUNT(*valuelist*)

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um DATASET ou de um conjunto de registros derivado de algumas condições de filtro, ou qualquer expressão que resulte em um conjunto de registros derivado, ou um nome de uma declaração DICTIONARY . Também pode ser a palavra-chave GROUP para indicar a contagem do número de elementos em cada grupo, quando usada em uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>expression</i>	Opcional. Uma expressão lógica indicando quais registros devem ser incluídos na contagem. Válido apenas quando o parâmetro recordset for a palavra-chave GROUP para indicar a contagem do número de elementos em um grupo.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
<i>valuelist</i>	Uma lista delimitada por vírgula das expressões a serem contadas. Também pode ser um SET de valores.
Return:	COUNT retorna um único valor.

A função **COUNT** retorna o número de registros no *recordset* ou *valuelist* especificado.

Exemplo:

```
MyCount := COUNT(Trades(Trades.trd_rate IN ['3', '4', '5']));
// count the number of records in the Trades record
// set whose trd_rate field contains 3, 4, or 5
R1 := RECORD
  person.per_st;
  person.per_sex;
  Number := COUNT(GROUP);
  //total in each state/sex category
  Hanks := COUNT(GROUP, person.per_first_name = 'HANK');
  //total of "Hanks" in each state/sex category
  NonHanks := COUNT(GROUP, person.per_first_name <> 'HANK');
  //total of "Non-Hanks" in each state/sex category
END;
T1 := TABLE(person, R1, per_st, per_sex);
```

```
Cnt1    := COUNT(4,8,16,2,1); //returns 5  
SetVals := [4,8,16,2,1];  
Cnt2    := COUNT(SetVals); //returns 5
```

Ver também: SUM, AVE, MIN, MAX, GROUP, TABLE

COVARIANCE

COVARIANCE(*recset*, *valuex*, *valuey* [, *expression*] [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Também pode ser a palavra-chave GROUP para indicar a operação nos elementos em cada grupo, quando usada em uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>valuex</i>	Um campo ou expressão numérica.
<i>valuey</i>	Um campo ou expressão numérica.
<i>expression</i>	Opcional. Uma expressão lógica indicando quais registros devem ser incluídos no cálculo. Válido apenas quando o parâmetro recset for a palavra-chave GROUP.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	COVARIANCE retorna um valor REAL único.

A função **COVARIANCE** retorna a extensão em que o *valuex* e *valuey* co-vary.

Exemplo:

```
pointRec := { REAL x, REAL y };
analyse( ds ) := MACRO
#uniqueName(stats)
%stats% := TABLE(ds, { c      := COUNT(GROUP),
    sx    := SUM(GROUP, x),
    sy    := SUM(GROUP, y),
    sxx   := SUM(GROUP, x * x),
    sxy   := SUM(GROUP, x * y),
    syy   := SUM(GROUP, y * y),
    varx  := VARIANCE(GROUP, x);
    vary  := VARIANCE(GROUP, y);
    varxy := COVARIANCE(GROUP, x, y);
    rc    := CORRELATION(GROUP, x, y) });
OUTPUT(%stats%);

// Following should be zero
OUTPUT(%stats%, { varx - (sxx-sx*sx/c)/c,
    vary - (syy-sy*sy/c)/c,
```

```
    varxy - (sxy-sx*sy/c)/c,  
    rc - (varxy/SQRT(varx*varxy)) }));  
  
OUTPUT(%stats%, { 'bestFit: y=' +  
    (STRING)((sy-sx*varxy/varx)/c) +  
    ' + ' +  
    (STRING)(varxy/varx)+'x' }));  
ENDMACRO;  
  
ds1 := DATASET([ {1,1},{2,2},{3,3},{4,4},{5,5},{6,6} ], pointRec);  
  
ds2 := DATASET([ {1.93896e+009, 2.04482e+009},  
    {1.77971e+009, 8.54858e+008},  
    {2.96181e+009, 1.24848e+009},  
    {2.7744e+009, 1.26357e+009},  
    {1.14416e+009, 4.3429e+008},  
    {3.38728e+009, 1.30238e+009},  
    {3.19538e+009, 1.71177e+009} ], pointRec);  
  
ds3 := DATASET([ {1, 1.00039},  
    {2, 2.07702},  
    {3, 2.86158},  
    {4, 3.87114},  
    {5, 5.12417},  
    {6, 6.20283} ], pointRec);  
  
analyse(ds1);  
analyse(ds2);  
analyse(ds3);
```

Ver também: VARIANCE, CORRELATION

CRON

CRON(*time*)

<i>time</i>	Uma expressão da string de caracteres contendo um tempo cron padrão unix.
Return:	CRON define um único evento de controladores de tempo.

A função **CRON** determina um evento temporal para uso no serviço **WHEN** ou função **WAIT**. Essa função é um sinônimo de **EVENT**('CRON', *time*).

O parâmetro de tempo segue o formato padrão de uma cron no unix, expressado em UTC (também conhecido como Tempo Médio de Greenwich) como uma string que contém os seguintes componentes de espaço delimitado:

minute hour dom month dow

<i>minute</i>	Um valor inteiro que significa o minuto da hora. Valores válidos de 0 a 59.
<i>hour</i>	Um valor inteiro que constitui a hora. Valores válidos de 0 a 23 (usando o relógio de 24 horas).
<i>dom</i>	Um valor inteiro que representa o dia do mês. Valores válidos de 1 a 31.
<i>month</i>	Um valor inteiro que representa o mês. Valores válidos de 1 a 12.
<i>dow</i>	Um valor inteiro que representa o dia da semana. Valores válidos de 0 a 6 (onde 0 representa o domingo).

Qualquer componente de *tempo* que você optar por não especificar, será substituído por um asterisco (*). Os intervalos de tempo devem ser definidos por um traço (-), as listas por uma vírgula (,), e “uma vez a cada n” usando a barra (/). Por exemplo, 6-18/3 no campo de hora acionará o controlador de tempo a cada três horas entre 6 da manhã e 6 da tarde, e 18-21/3,0-6/3 acionará o controlador de tempo a cada três horas entre 6 da tarde e 6 da manhã.

Exemplo:

```
EXPORT events := MODULE
  EXPORT dailyAtMidnight := CRON('0 0 * * *');
  EXPORT dailyAt( INTEGER hour,
    INTEGER minute=0 ) :=
    EVENT('CRON',
      (STRING)minute + ' ' + (STRING)hour + ' * * * ');
  EXPORT dailyAtMidday := dailyAt(12, 0);
  EXPORT EveryThreeHours := CRON('0 0-23/3 * * *');
END;

BUILD(teenagers) : WHEN(events.dailyAtMidnight);
BUILD(oldies)    : WHEN(events.dailyAt(6));
BUILD(NewStuff)  : WHEN(events.EveryThreeHours);
```

Ver também: **EVENT**, **WHEN**, **WAIT**, **NOTIFY**

DEDUP

DEDUP(*recordset* [, *condition* [[**MANY**], **ALL**[, **HASH**]] [,**BEST** (*sort-list*)] [, **KEEP** *n*] [, *keeper*]] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros a ser processado, normalmente classificado na mesma ordem em que a expressão será testada. Pode ser o nome de um dataset ou de um record set derivado, ou qualquer expressão que resulte em um conjunto de registro derivado.
<i>condition</i>	Opcional. Uma lista delimitada por vírgula de expressões ou campos principais no record set que define os registros “duplicados”. As palavras-chave LEFT e RIGHT podem ser usadas como qualificadores de dataset nos campos do recordset. Se a condição for omitida, cada campo de record set se tornará uma condição de correspondência. As palavras-chave RECORD (ou WHOLE RECORD) podem ser usadas para indicar todos os campos nessa estrutura, e/ou você pode usar a palavra-chave EXCEPT para listar os campos de não deduplicação na estrutura.
MANY	Opcional. Especifica ou desempenha uma classificação/deduplicação local antes de localizar conteúdos duplicados globalmente. Isso é muito mais útil quando houver expectativa de vários conteúdos duplicados.
ALL	Opcional. Faz a correspondência da condição em relação a todos os registros, e não apenas em relação aos registros adjacentes. Esta opção pode mudar a ordem do resultado dos registros resultantes.
HASH	Opcional. Especifica que a operação ALL é realizada com o uso de tabelas de hash.
BEST	Opcional. Oferece controle adicional sobre quais registros são mantidos de um conjunto de registros “duplicados”. O primeiro na ordem de registros <sort-list> será mantido. BEST não pode ser usado com um parâmetro KEEP maior do que 1.
<i>sort-list</i>	Uma lista de campos delimitada por vírgula que define os registros duplicados a serem mantidos. Os campos podem ser prefixados com um sinal de subtração para solicitar uma classificação reversa desse campo.
KEEP	Opcional. Especifica manter <i>n</i> números de registros duplicados. Se omitido, o comportamento padrão é KEEP 1. Não é válido se a opção ALL estiver presente.
<i>n</i>	O número de registros duplicados a serem mantidos.
<i>keeper</i>	Opcional. As palavras-chave LEFT ou RIGHT LEFT (padrão, se omitido) mantém o primeiro registro encontrado e RIGHT mantém o último.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.

Referência a Linguagem ECL

Ações e Funções Built-in

<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	DEPUP retorna um conjunto de registros.

A função **DEDUP** analisa o *recordset* para encontrar registros duplicados, como definido pelo parâmetro da *condição*, e retorna um conjunto de retorno único. Essa função é semelhante à declaração DISTINCT na declaração SQL. O *recordset* deve ser classificado, a menos que ALL tenha sido especificada.

Se um parâmetro de *condição* for um valor único (*field*), DEDUP realiza uma deduplicação simples em nível de campo equivalente a LEFT *field*=RIGHT *field*. A *condição* é avaliada para cada par de registros adjacentes no record set. Se a *condição* retornar TRUE, o registro *detentor* será mantido e o outro removido.

A opção **ALL** significa que cada par de registro é avaliado – em vez de apenas aqueles adjacentes uns aos outros – independentemente da ordem de classificação. A avaliação é tal que, para os registros 1, 2, 3, 4, os pares de registros que são comparados entre si são:

(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)

Isso significa que são realizadas suas comparações para cada par, permitindo que a *condição* seja não cumulativa.

KEEP *n* significa efetivamente deixar *n* registros de cada tipo duplicado. Isso é útil para amostragem. O valor **detentor** LEFT (implícito se LEFT ou RIGHT não tiverem sido especificados) significa que se os registros left e right atendem aos critérios de deduplicação (ou seja, eles “coincidem”), o registro left é mantido. Mas se em vez disso o **detentor** RIGHT aparecer, o right será mantido. Em ambos os casos, a próxima comparação envolve o sobrevivente da deduplicação; dessa forma, vários registros duplicados podem se transformar em apenas um.

A opção **BEST** oferece controle adicional sobre quais registros são mantidos de um conjunto de registros “duplicados”. O primeiro na ordem de registros *sort-list* será mantido. A *sort-list* é uma lista de campos delimitada por vírgula. Os campos podem ser prefixados com um sinal de subtração para solicitar uma classificação reversa desse campo.

DEDUP(recordset, field1, BEST(field2)) significa que nos registros duplicados, o primeiro registro do conjunto de duplicados classificado por field2 será mantido. DEDUP(recordset, field1, BEST(-field2)) gera o último registro classificado por field2 no conjunto de duplicados.

A opção BEST não pode ser usada com um parâmetro KEEP maior do que 1.

Exemplo:

```
SomeFile := DATASET([{'001','KC','G'},
                    {'002','KC','Z'},
                    {'003','KC','Z'},
                    {'004','KC','C'},
                    {'005','KA','X'},
                    {'006','KB','A'},
                    {'007','KB','G'},
                    {'008','KA','B'}],{STRING3 Id, String2 Value1, String1 Value2});

SomeFile1 := SORT(SomeFile, Value1);

DEDUP(SomeFile1, Value1, BEST(Value2));
// Output:
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C

DEDUP(SomeFile1, Value1, BEST(-Value2));
// Output:
// id value1 value2
```

```
// 005 KA X
// 007 KB G
// 002 KC Z

DEDUP(SomeFile1, Value1, HASH, BEST(Value2));
// Output:
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C
```

Condições complexas de Record Set

A função DEDUP com a opção ALL é útil para determinar as condições complexas do recordset entre os registros de um mesmo conjunto. Embora DEDUP seja tradicionalmente usado para eliminar registros duplicados próximos uns aos outros no record set, a expressão condicional combinada com a opção ALL estende esta capacidade. A opção ALL faz com que cada registro seja comparado de acordo com a expressão condicional para cada um dos demais registros do conjunto de registros. Este recurso é mais eficaz com conjuntos de registros pequenos; os conjuntos maiores também devem usar a opção HASH.

Exemplo:

```
LastTbl := TABLE(Person, {per_last_name});
Lasts   := SORT>LastTbl, per_last_name);
MySet   := DEDUP>Lasts, per_last_name);
        // unique last names -- this is exactly equivalent to:
        //MySet := DEDUP>Lasts, LEFT.per_last_name=RIGHT.per_last_name);
        // also exactly equivalent to:
        //MySet := DEDUP>Lasts);

NamesTbl1 := TABLE(Person, {per_last_name, per_first_name});
Names1    := SORT>NamesTbl1, per_last_name, per_first_name);
MyNames1  := DEDUP>Names1, RECORD);
        //dedup by all fields -- this is exactly equivalent to:
        //MyNames1 := DEDUP>Names, per_last_name, per_first_name);
        // also exactly equivalent to:
        //MyNames1 := DEDUP>Names1);

NamesTbl2 := TABLE(Person, {per_last_name, per_first_name, per_sex});
Names2    := SORT>NamesTbl2, per_last_name, per_first_name);
MyNames2  := DEDUP>Names, RECORD, EXCEPT per_sex);
        //dedup by all fields except per_sex
        // this is exactly equivalent to:
        //MyNames2 := DEDUP>Names, EXCEPT per_sex);

/* In the following example, we want to determine how many 'AN' or 'AU' type inquiries
have occurred within 3 days of a 'BB' type inquiry.
The COUNT of inquiries in the deduped recordset is subtracted from the COUNT
of the inquiries in the original recordset to provide the result.*/
INTEGER abs(INTEGER i) := IF ( i < 0, -i, i );
WithinDays(ldrpt, lday, rdrpt, rday, days) :=
    abs(DaysAgo(ldrpt, lday)-DaysAgo(rdrpt, rday)) <= days;
DedupedInqs := DEDUP>(inquiry, LEFT.inq_ind_code='BB' AND
    RIGHT.inq_ind_code IN ['AN', 'AU'] AND
        WithinDays(LEFT.inq_drpt,
            LEFT.inq_drpt_day,
            RIGHT.inq_drpt,
            RIGHT.inq_drpt_day, 3),
    ALL );
InqCount := COUNT>(Inquiry) - COUNT>(DedupedInqs);
OUTPUT>(person(InqCount >0), {InqCount});
```

Ver também: SORT, ROLLUP, TABLE, Estrutura FUNCTION

DEFINE

DEFINE(*pattern*, *symbol*)

<i>pattern</i>	O nome de um padrão de análise RULE .
<i>symbol</i>	Uma constante da string que especifica o nome a ser usado na opção USE em uma função PARSE ou a função USE em um padrão de análise RULE .
Return:	DEFINE cria um padrão RULE.

A função **DEFINE** define um *símbolo* para o *padrão* especificado que pode ser encaminhado para referência em atributos de padrão de análise previamente definidos. Esse é o único tipo de referência de encaminhamento de permitido no ECL.

Exemplo:

```
RULE a := USE('symbol');  
  //uses the 'symbol'pattern defined later - b  
RULE b := 'pattern';  
  //defines a rule pattern  
RULE s := DEFINE(b,'symbol');  
  //associate the "b" rule with the  
  //'symbol' for forward reference by rule "a
```

Ver também: PARSE, PARSE Pattern Value Types

DENORMALIZE

DENORMALIZE(*parentrecset*, *childrecset*, *condition*, *transform* [, **LOCAL**] [, **NOSORT**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

DENORMALIZE(*parentrecset*, *childrecset*, *condition*, **GROUP**, *transform* [, **LOCAL**] [, **NOSORT**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>parentrecset</i>	O conjunto de registros primários a ser processado, já no formato que conterá os registros primários e secundários desnormalizados.
<i>childrecset</i>	O conjunto de registros secundários a ser processado.
<i>condition</i>	Uma expressão que especifica como corresponder registros entre o <i>parentrecset</i> e <i>childrecset</i> .
<i>transform</i>	A função TRANSFORM a ser acionada.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
NOSORT	Opcional. Especifica que a operação é desempenhada sem classificar <i>parentrecset</i> ou <i>childrecset</i> – ambos já devem ter sido classificados para que os registros de correspondência de ambos estejam em ordem. Isso permite que o programador controle a ordem dos registros secundários.
GROUP	Especifica o agrupamento de registros <i>childrecset</i> com base na condição JOIN para que todos os registros secundários relacionados sejam especificados para o TRANSFORM como um parâmetro de dataset.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	DENORMALIZE retorna um conjunto de registros.

A função **DENORMALIZE** é usada para formar um registro combinado a partir de um registro primário e de qualquer número de registros secundários. Ela atua de forma bastante semelhante ao JOIN, exceto onde JOIN com um registro primário e três secundários acionaria *transform* três vezes e geraria três resultados, o DENORMALIZE aciona *transform* três vezes, sendo que a entrada para o primeiro *transform* consiste de um registro primário e um secundário, a entrada para o segundo *transform* consiste do resultado do primeiro *transform* e outro registro secundário, e a entrada para o terceiro *transform* constitui do resultado do segundo *transform* e o registro secundário restante. Além disso, assim como JOIN, a ordem na qual os registros *childrecset* são enviados para *transform* é indefinida.

Uma vez que DENORMALIZE é basicamente uma forma especializada de JOIN, os diversos tipos de join (LEFT OUTER, RIGHT OUTER, FULL OUTER, LEFT ONLY, RIGHTONLY, FULL ONLY) também podem ser usados no DENORMALIZE e atuam da mesma forma que quando são usados no JOIN.

Todas as opções JOIN estão disponíveis para o DENORMALIZE. Veja Opções de join para obter mais informações.

Requerimentos da Função DENORMALIZE TRANSFORM

Para a forma 1, a função *transform* precisa adotar pelo menos dois parâmetros: o registro LEFT de mesmo formato que a combinação de *parentrecset* e *childrecset* (a estrutura resultante do registro desnormalizado), e um registro RIGHT de mesmo formato que *childrecset*. Um terceiro parâmetro opcional pode ser especificado: um COUNTER inteiro especificando o número de vezes que *transform* foi acionado para o conjunto atual de pares primários/secundários (definido pelos valores da *condição*). O resultado da função *transform* deve ser um record set de mesmo formato que o registro LEFT.

Para a forma 2, a função *transform* precisa adotar pelo menos dois parâmetros: o registro LEFT de mesmo formato que a combinação de *parentrecset* e *childrecset* (a estrutura resultante do registro desnormalizado), e um dataset ROWS(RIGHT) de mesmo formato que *childrecset*. O resultado da função *transform* deve ser um record set de mesmo formato que o registro LEFT.

Exemplo:

Exemplo da forma 1:

```
NormRec := RECORD
  STRING20  thename;
  STRING20  addr;
END;
NamesRec := RECORD
  UNSIGNED1  numRows;
  STRING20  thename;
  STRING20  addr1 := '';
  STRING20  addr2 := '';
  STRING20  addr3 := '';
  STRING20  addr4 := '';
END;
NamesTable := DATASET([ {0,'Kevin'}, {0,'Liz'}, {0,'Mr Nobody'},
                        {0,'Anywhere'}], NamesRec);
NormAddrs := DATASET([ {'Kevin','10 Malt Lane'},
                        {'Liz','10 Malt Lane'},
                        {'Liz','3 The cottages'},
                        {'Anywhere','Here'},
                        {'Anywhere','There'},
                        {'Anywhere','Near'},
                        {'Anywhere','Far'}], NormRec);
NamesRec DeNormThem(NamesRec L, NormRec R, INTEGER C) := TRANSFORM
  SELF.NumRows := C;
  SELF.addr1 := IF (C=1, R.addr, L.addr1);
  SELF.addr2 := IF (C=2, R.addr, L.addr2);
  SELF.addr3 := IF (C=3, R.addr, L.addr3);
  SELF.addr4 := IF (C=4, R.addr, L.addr4);
  SELF := L;
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
                             LEFT.thename = RIGHT.thename,
                             DeNormThem(LEFT,RIGHT,COUNTER));
OUTPUT(DeNormedRecs);
```

Exemplo da forma 2:

```
NormRec := RECORD
  STRING20  thename;
  STRING20  addr;
END;
NamesRec := RECORD
```

```
UNSIGNED1 numRows;
STRING20 thename;
DATASET(NormRec) addresses;
END;
NamesTable := DATASET([ {0,'Kevin',[ ]},{0,'Liz',[ ]},
                        {0,'Mr Nobody',[ ]},{0,'Anywhere',[ ]}],
                      NamesRec);
NormAddrs := DATASET([ {'Kevin','10 Malt Lane'},
                      {'Liz','10 Malt Lane'},
                      {'Liz','3 The cottages'},
                      {'Anywhere','Here'},
                      {'Anywhere','There'},
                      {'Anywhere','Near'},
                      {'Anywhere','Far'}],NormRec);
NamesRec DeNormThem(NamesRec L, DATASET(NormRec) R) := TRANSFORM
    SELF.NumRows := COUNT(R);
    SELF.addresses := R;
    SELF := L;
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
                            LEFT.thename = RIGHT.thename,
                            GROUP,
                            DeNormThem(LEFT,ROWS(RIGHT)));
OUTPUT(DeNormedRecs);
```

Exemplo NOSORT:

```
ParentFile.Value1;
ParentFile.Value2;
STRING1 CVal2_1 := '';
STRING1 CVal2_2 := '';
END;
P_Recs := TABLE(ParentFile, MyOutRec);
MyOutRec DeNormThem(MyOutRec L, MyRec R, INTEGER C) := TRANSFORM
    SELF.CVal2_1 := IF(C = 1, R.Value2, L.CVal2_1);
    SELF.CVal2_2 := IF(C = 2, R.Value2, L.CVal2_2);
    SELF := L;
END;
DeNormedRecs := DENORMALIZE(P_Recs, ChildFile,
                            LEFT.Value1 = RIGHT.Value1,
                            DeNormThem(LEFT,RIGHT,COUNTER),NOSORT);
OUTPUT(DeNormedRecs);
/* DeNormedRecs result set is:
Rec#  Value1 PVal2  CVal2_1  CVal2_2
1      A      C      Z      T
2      B      B      S      Y
3      C      A      X      W
*/
```

Ver também: JOIN, Estrutura TRANSFORM, Estrutura RECORD, NORMALIZE

DISTRIBUTE "Randômico"

DISTRIBUTE(*recordset* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

DISTRIBUTE(*recordset*, *expression* [, **MERGE**(*sorts*)] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

DISTRIBUTE(*recordset*, *index* [, *joincondition*] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

DISTRIBUTE(*recordset*, **SKEW**(*maxskew* [, *skewlimit*]) [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros a ser distribuído.
<i>expression</i>	Uma expressão inteira que especifica como distribuir o conjunto de registros, geralmente usando uma das funções HASH para fins de eficiência.
MERGE	Especifica que os dados são redistribuídos, mantendo a ordem de classificação local em cada nó.
<i>sorts</i>	As expressões de classificação pelas quais os dados foram localmente classificados.
<i>index</i>	O nome da definição de um atributo INDEX , que fornece a distribuição adequada.
<i>joincondition</i>	Opcional. Uma expressão lógica que especifica como vincular os registros ao conjunto de registros e índice. As palavras-chave LEFT e RIGHT podem ser usadas como qualificadores de dataset nos campos do recordset e do índice.
SKEW	Especifica os valores de distorção de dados permitidos.
<i>maxskew</i>	Um número de ponto flutuante no intervalo de zero (0,0) a um (1,0) especificando a distorção mínima a ser permitida (0,1=10%).
<i>skewlimit</i>	Opcional. Um número de ponto flutuante no intervalo de zero (0,0) a um (1,0) especificando a distorção máxima a ser permitida (0,1=10%).
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for "False" (Falso), especifica que a ordem do registro de resultado não é importante. Quando for "True" (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	DISTRIBUTE retorna um conjunto de registros.

A função **DISTRIBUTE** redistribui registros do *recordset* para todos os nós do cluster.

"Random" DISTRIBUTE

DISTRIBUTE(*recordset*)

Esta forma redistribui o *recordset* "aleatoriamente" para que não haja distorção de dados entre os nós, porém sem as desvantagens que a função RANDOM() poderia introduzir. Isso é funcionalmente equivalente a distribuição do registro inteiro através de uma hash.

Expressão DISTRIBUTE

DISTRIBUTE(*recordset*, *expression*)

Esta forma redistribui o *recordset* com base na *expressão especificada*, normalmente uma das funções HASH . Somente os 32 bits inferiores do valor da *expressão* são usados; portanto, HASH ou HASH32 são a escolha ideal. Os registros para os quais a *expressão* apresenta o mesmo resultado que estão no mesmo nó. DISTRIBUTE executa implicitamente uma operação de módulo se o valor da *expressão* não estiver no intervalo do número de nós disponíveis.

Se a opção MERGE for especificada, o *recordset* precisa ter sido classificado localmente pelas expressões *sort* . Isso evita uma reclassificação.

DISTRIBUTE baseado em Index

DISTRIBUTE(*recordset*, *index* [, *joincondition*])

Esta forma redistribui o *recordset* com base na distribuição existente do *index* especificado, onde a ligação entre os dois é determinada pela *joincondition*. Os registros para os quais a *joincondition* é true (verdadeira) terminarão no mesmo nó.

DISTRIBUTE baseado em Skew

DISTRIBUTE(*recordset*, **SKEW**(*maxskew* [, *skewlimit*]))

Esta forma redistribui o *recordset* , mas apenas se for necessário. A finalidade desta forma é substituir o uso de DISTRIBUTE(*recordset*,RANDOM()) para apenas obter uma distribuição relativamente uniforme dos dados entre os nós. Esta forma sempre tentará minimizar a quantidade de dados redistribuídos entre os nós.

A distorção de um dataset é calculada da seguinte forma:

$\text{MAX}(\text{ABS}(\text{AvgPartSize} - \text{PartSize}[\text{node}]) / \text{AvgPartSize})$

Se a distorção do *recordset* for menor do que a do *maxskew* , então DISTRIBUTE será não operacional. Se o *limite de distorção* for especificado e se a distorção em qualquer nó exceder esse limite, a tarefa falhará e exibirá uma mensagem de erro (especificando o número do primeiro nó que excedeu o limite); caso contrário, os dados são redistribuídos para garantir que os dados sejam distribuídos com menor distorção que *maxskew*.

Exemplo:

```
MySet1 := DISTRIBUTE(Person); //"random" distribution - no skew
MySet2 := DISTRIBUTE(Person,HASH32(Person.per_ssn));
//all people with the same SSN end up on the same node
//INDEX example:
mainRecord := RECORD
  INTEGER8 sequence;
  STRING20 forename;
  STRING20 surname;
  UNSIGNED8 filepos{VIRTUAL(fileposition)};
END;
mainTable := DATASET('~keyed.d00',mainRecord,THOR);
nameKey := INDEX(mainTable, {surname,forename,filepos}, 'name.idx');
incTable := DATASET('~inc.d00',mainRecord,THOR);
x := DISTRIBUTE(incTable, nameKey,
```

```
        LEFT.surname = RIGHT.surname AND
        LEFT.forename = RIGHT.forename);
OUTPUT(x);

//SKEW example:
Jds := JOIN(somedata,otherdata,LEFT.sysid=RIGHT.sysid);
Jds_dist1 := DISTRIBUTE(Jds,SKEW(0.1));
//ensures skew is less than 10%
Jds_dist2 := DISTRIBUTE(Jds,SKEW(0.1,0.5));
//ensures skew is less than 10%
//and fails if skew exceeds 50% on any node
```

Ver também: HASH32, DISTRIBUTED, INDEX

DISTRIBUTED

DISTRIBUTED(*recordset* [, *expression*])

<i>recordset</i>	O conjunto de registros distribuídos.
<i>expression</i>	Opcional. Uma expressão que especifica como o conjunto de registros é distribuído.
Return:	DISTRIBUTED retorna um conjunto de registros.

A função **DISTRIBUTED** é uma diretiva de compilador indicando que os registros do *recordset* já foram distribuídos entre os nós da Refinaria de Dados com base na *expressão* especificada. Os registros para os quais a *expressão* avalia a mesma estão no mesmo nó.

Se a *expressão* se a expressão for omitida, a função simplesmente oculta um aviso que é às vezes gerado informando que o *recordset* não foi distribuído.

Exemplo:

```
MySet := DISTRIBUTED(Person, HASH32(Person.per_ssn));  
//all people with the same SSN are already on the same node
```

Ver também: HASH32, DISTRIBUTE

DISTRIBUTION

DISTRIBUTION(*recordset* [, *fields*] [, **NAMED**(*name*)] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros no qual as estatísticas serão executadas.
<i>fields</i>	Opcional. Uma lista de campos delimitada por vírgula no conjunto de registros para o qual a ação será limitada. Se omitido, todos os campos serão incluídos.
NAMED	Opcional. Especifica o nome do resultado que aparece na workunit.
<i>name</i>	Uma constante de string que contém o rótulo do resultado.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

A ação **DISTRIBUTION** gera um relatório de tabela de referência cruzada em formato XML indicando quantos registros exclusivos existem no *recordset* para cada valor em cada campo deste *recordset*.

Quando há um número excessivamente amplo de valores distintos, a ação retorna uma estimativa na seguinte forma:

```
<XML>
  <Field name="seqnum" estimate="4000000" />
</XML>
```

O tipo de dado DECIMAL não é suportado por esta ação. Alternativamente, você pode usar o tipo de dados REAL.

Exemplo:

```
SomeFile := DATASET([{'C','G'},{'C','C'},{'A','X'},{'B','G'}],
  {STRING1 Value1,STRING1 Value2});
DISTRIBUTION(SomeFile);
/* The result comes back looking like this:
<XML>
<Field name="Value1" distinct="3">
  <Value count="1">A</Value>
  <Value count="1">B</Value>
  <Value count="2">C</Value>
</Field>
<Field name="Value2" distinct="3">
  <Value count="1">C</Value>
  <Value count="2">G</Value>
  <Value count="1">X</Value>
</Field>
</XML>
*/
```

```
//*****
namesRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  INTEGER2 age;
END;

namesTable := DATASET([
  {'Halligan', 'Kevin', 31},
  {'Halligan', 'Liz', 30},
  {'Salter', 'Abi', 10},
  {'X', 'Z', 5}], namesRecord);

DISTRIBUTION(namesTable, surname, forename, NAMED('Stats'));
/* The result comes back looking like this:
<XML>
<Field name="surname" distinct="3">
  <Value count="2">Halligan</Value>
  <Value count="1">X</Value>
  <Value count="1">Salter</Value>
</Field>
<Field name="forename" distinct="4">
  <Value count="1">Abi</Value>
  <Value count="1">Kevin</Value>
  <Value count="1">Liz</Value>
  <Value count="1">Z</Value>
</Field>
</XML>
*/

//Post-processing the result with PARSE:
x := DATASET(ROW(TRANSFORM({STRING line},
  SELF.line := WORKUNIT('Stats', STRING))));
res := RECORD
  STRING Fieldname := XMLTEXT('@name');
  STRING Cnt := XMLTEXT('@distinct');
END;

out := PARSE(x, line, res, XML('XML/Field'));
out;
```

EBCDIC

EBCDIC(*recordset* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os threads <i>numthreads</i> . <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	EBCDIC retorna um conjunto de registros.

..

A função **ASCII** retorna o *recordset* com todos os campos da **STRING** traduzidos do EBCDIC para ASCII.

Exemplo:

```
EBCDICRecs := EBCDIC(SomeASCIIInput);
```

Ver também: **ASCII**

ENTH

ENTH(*recordset*, *numerator* [, *denominator* [, *which*]] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para amostragem. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
<i>numerator</i>	O número de registros a ser retornado. Os registros selecionados são espaçados por todo o conjunto de registros.
<i>denominator</i>	Opcional. O tamanho de cada conjunto a partir do qual o número do numerador de registros será retornado. Se omitido, o valor do denominador será o número total de registros no recordset.
<i>which</i>	Opcional. Um número inteiro que especifica o número ordinal do conjunto de amostra a ser retornado. Isso é usado para obter múltiplas amostras sem sobreposição a partir de um mesmo conjunto de registros. Se o numerador não for igual a 1, alguns registros podem ser sobrepostos.
LOCAL	Opcional. Especifica que a amostra é extraída em cada nó do supercomputador sem considerar o número de registros nos demais nós, melhorando significativamente o desempenho se os resultados exatos não forem obrigatórios.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	ENTH retorna um conjunto de registros.

A função **ENTH** retorna uma amostra de conjunto de registros a partir do *recordset* nominado. ENTH retorna o número do *numerador* de registros de cada record set do *denominator* no *recordset*. A menos que LOCAL seja especificado, os registros são escolhidos nos intervalos exatos em todos os nós do supercomputador.

Exemplo:

```
MySample1 := ENTH(Person,1,10,1); // 10% (1 out of every 10)
MySample2 := ENTH(Person,15,100,1); // 15% (15 out of every 100)
MySample3 := ENTH(Person,3,4,1); // 75% (3 out of every 4)

SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                    {'F'},{'G'},{'H'},{'I'},{'J'},
                    {'K'},{'L'},{'M'},{'N'},{'O'},
                    {'P'},{'Q'},{'R'},{'S'},{'T'},
                    {'U'},{'V'},{'W'},{'X'},{'Y'}],
                    {STRING1 Letter});
Set1 := ENTH(SomeFile,2,10,1); // returns E, J, O, T, Y
```

See Also: CHOOSEN, SAMPLE

ERROR

ERROR [(*errormessage* | *errorcode*)] ;

ERROR (*errorcode* , *errormessage*) ;

ERROR(*datatype* [, [*errorcode*] [, *errormessage*]]) ;

<i>errormessage</i>	Opcional. Uma constante da string que contém a mensagem a ser exibida.
<i>errorcode</i>	Opcional. Uma constante de número inteiro que contém o número do erro a ser exibido.
<i>datatype</i>	O tipo de valor ou nome de uma estrutura RECORD . Pode usar a função TYPEOF.

A função **ERROR** interrompe imediatamente o processamento na tarefa e exibe o *errorcode* e/ou a *errormessage*. A terceira forma está disponível para ser usada em contextos onde é exigido um tipo de valor ou de dataset. Esta função faz o mesmo que a ação FAIL, mas pode ser usada em um contexto da expressão, tal como em uma função TRANSFORM .

Exemplo:

```
outrec Xform(inrec L, inrec R) := TRANSFORM
  SELF.key := IF(L.key <= R.key, R.key, ERROR('Recs not in order'));
END;
```

Ver também: FAILURE, FAIL

EVALUATE

EVALUATE action

[attrname :=] EVALUATE(expression) ;

[attrname :=] EVALUATE(module [, defname]) ;

<i>attrname</i>	Opcional. O nome da ação, que transforma a ação em uma definição, consequentemente não é executado até que <i>attrname</i> seja usado como uma ação.
<i>expression</i>	A função a ser acionada em um contexto de ação.
<i>module</i>	O módulo a ser avaliado.
<i>defname</i>	Opcional. O nome de uma definição específica dentro do <i>módulo</i> a ser avaliado. Se omitido, todas as definições no <i>módulo</i> serão avaliadas.

A primeira forma da ação **EVALUATE** nomeia uma *expressão* (tipicamente o acionamento de uma função) a ser executada em um contexto de ação. Isso é útil principalmente ao acionar funções que tenham efeitos colaterais, quando o valor do retorno não é importante.

A segunda forma da ação **EVALUATE** recursivamente expande e avalia as definições exportadas do *módulo* . Se um *defname* for especificado, apenas essa definição será avaliada.

Exemplo:

Exemplo da Form 1:

```
myService := SERVICE
  UNSIGNED4 doSomething(String text);
END;

ds := DATASET('MyFile', {STRING20 text} , THOR);

APPLY(ds, EVALUATE(doSomething(ds.text)));
//calls the doSomething function once for each record in the ds
// dataset, ignoring the returned values from the function
```

Exemplo de Form 2:

```
M := MODULE
  EXPORT a := 10;
  EXPORT b := OUTPUT('Hello');
END;

M2 := MODULE
  EXPORT mx := M;
  EXPORT d := OUTPUT('Richard');
END;

EVALUATE(M2);
//produces three results:
// Result_1: 10
// Result_2: Hello
// Result_3: Richard
```

See Also: APPLY, SERVICE Structure,

Função EVALUATE

EVALUATE(*onerecord*, *value*)

<i>onerecord</i>	Um conjunto de registros que consiste de um único registro.
<i>value</i>	O valor a ser retornado. Isso pode ser qualquer expressão que produza um valor.
Return:	EVALUATE retorna um único valor.

A função **EVALUATE** retorna o *valor* avaliado no contexto do conjunto *onerecord* (o qual deve ser apenas um único registro). Esta função normalmente usa a indexação para selecionar um único registro para o record set *onerecord*. O uso serve para retornar um valor de um registro secundário específico quando estiver operando no nível de escopo do registro primário. A vantagem de EVALUATE sobre o uso da indexação do record set em um único campo é que o *valor* retornado pode ser constituído de qualquer expressão, e não apenas de um único campo do dataset secundário (child dataset).

Acessando dados em nível de campo em um Registro Específico

Para acessar os dados de nível de campo em um registro específico, os recursos de indexação do conjunto de registro deve ser usado para selecionar um único registro. A função SORT e os filtros do conjunto de registros são úteis na seleção e ordenação do conjunto de registros para que o registro adequado possa ser selecionado.

Exemplo:

```
WorstCard := SORT(Cards,Std.Scoring);
MyValue   := EVALUATE(WorstCard[1],Std.Utilization);
// WorstCard[1] uses indexing to get the first record
// in the sort order, then evaluates that record
// returning the Std.Utilization value

ValidBalTrades := trades(ValidMoney(trades.trd_bal));
HighestBals := SORT(ValidBalTrades,-trades.trd_bal);
Highest_HC := EVALUATE(HighestBals[1],trades.trd_hc);
//return trd_hc field of the trade with the highest balance
// could also be coded as (using indexing):
// Highest_HC := HighestBals[1].trades.trd_hc;

OUTPUT(Person,{per_last_name,per_first_name,Highest_HC});
//output that Highest_HC for each person
//This output operates at the scope of the Person record
// EVALUATE is needed to get the value from a Trades record
// because Trades is a Child of Person

IsValidInd := trades.trd_ind_code IN ['FM','RE'];
IsMortgage := IsValidInd OR trades.trd_rate = 'G';
SortedTrades := SORT(trades(ValidDate(trades.trd_dopn),isMortgage),
    trades.trd_dopn_mos);
CurrentRate := MAP(~EXISTS(SortedTrades) => ' ',
    EVALUATE(SortedTrades[1], trades.trd_rate));

OUTPUT(person,{CurrentRate});
```

See Also: SORT

EVENT

EVENT(*event* , *subtype*)

<i>event</i>	Uma constante de strings, com distinção entre maiúsculas e minúsculas, que nomeia o evento para interceptação.
<i>subtype</i>	Uma constante de strings, com distinção entre maiúsculas e minúsculas, que nomeia o tipo de evento específico para interceptação. Pode conter * e ? como correspondência-curinga do subtipo do evento.
Return:	EVENT retorna um único evento.

A função **EVENT** retorna um evento acionado, que pode ser usado no serviço do fluxo de trabalho **WHEN** ou na ação **NOTIFY** . ou nas ações **WAIT** e **NOTIFY**.

Exemplo:

```
IMPORT STD;
MyEventName := 'MyFileEvent';
MyFileName  := 'test::myfile';

IF (STD.File.FileExists(MyFileName),
    STD.File.DeleteLogicalFile(MyFileName));
//deletes the file if it already exists

STD.File.MonitorLogicalFileName(MyEventName,MyFileName);
//sets up monitoring and the event name
//to fire when the file is found

OUTPUT('File Created') : WHEN(EVENT(MyEventName,'*'),COUNT(1));
//this OUTPUT occurs only after the event has fired

afile := DATASET([{'A', '0'}], {STRING10 key,STRING10 val});
OUTPUT(afile,,MyFileName);
//this creates a file that the DFU file monitor will find
//when it periodically polls

//*****
EXPORT events := MODULE
  EXPORT dailyAtMidnight := CRON('0 0 * * *');
  EXPORT dailyAt( INTEGER hour,
    INTEGER minute=0) :=
    EVENT('CRON',
      (STRING)minute + ' ' + (STRING)hour + ' * * *');
  EXPORT dailyAtMidday := dailyAt(12, 0);
END;
BUILD(teenagers): WHEN(events.dailyAtMidnight);
BUILD(oldies)   : WHEN(events.dailyAt(6));
```

Ver também: **EVENTNAME**, **EVENTEXTRA**, **CRON**, **WHEN**, **WAIT**, **NOTIFY**

EVENTNAME

EVENTNAME

Return:	retorna um valor único da string.
---------	-----------------------------------

EVENTNAME retorna o nome do evento a ser desencadeado.

Exemplo:

```
doMyService := FUNCTION
  OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
  NOTIFY(EVENT('MyServiceComplete',
    '<Event><returnTo>FRED</returnTo></Event>'),
    EVENTEXTRA('returnTo'));
  RETURN EVENTEXTRA('returnTo');
END;

doMyService : WHEN('MyService');

// and a call
NOTIFY('MyService',
  '<Event><returnTo>' + WORKUNIT + '</returnTo></Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

Ver também: EVENT, EVENTEXTRA, CRON, WHEN, WAIT, NOTIFY

EVENTEXTRA

EVENTEXTRA(*tag*)

Return:	EVENTEXTRA retorna um valor único da string.
---------	--

A **função** EVENTEXTRA retorna o conteúdo da *tag* do texto XML no segundo parâmetro da função EVENT.

Exemplo:

```
doMyService := FUNCTION
  OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
  NOTIFY(EVENT('MyServiceComplete',
    '<Event><returnTo>FRED</returnTo></Event>'),
    EVENTEXTRA('returnTo'));
  RETURN EVENTEXTRA('returnTo');
END;

doMyService : WHEN('MyService');

// and a call
NOTIFY('MyService',
  '<Event><returnTo>' + WORKUNIT + '</returnTo></Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

Ver também: EVENT, EVENTNAME, CRON, WHEN, WAIT, NOTIFY

EXISTS

EXISTS(*recordset* [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL**] [(*numthreads*)] [, **ALGORITHM**(*name*)])

EXISTS(*valuelist*)

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um índice, de um dataset ou de um conjunto de registros derivado de algumas condições de filtro, ou qualquer expressão que resulte em um conjunto de registros derivado.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
<i>valuelist</i>	Uma lista delimitada por vírgula das expressões. Também pode ser um SET de valores.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	EXISTS retorna um único valor booleano.

A função **EXISTS** retorna como “true” (verdadeiro) se o número de registros no *recordset* especificado for > 0, ou o *valuelist* for preenchido. Isso é mais comumente usado para detectar se um filtro realizou a filtragem de todos os registros.

Ao verificar um recordset vazio, use a função **EXISTS**(*recordset*) em vez da expressão: **COUNT**(*recordset*) > 0. O uso de **EXISTS** resulta em um processamento mais eficiente e melhor desempenho sob essas circunstâncias.

Exemplo:

```
MyBoolean := EXISTS(Publics(pub_type = 'B'));
TradesExistPersons := Person(EXISTS(Trades));
NoTradesPerson := Person(NOT EXISTS(Trades));

MinVal2 := EXISTS(4,8,16,2,1); //returns TRUE
SetVals := [4,8,16,2,1];
MinVal3 := EXISTS(SetVals); //returns TRUE
NullSet := [];
MinVal3 := EXISTS(NullSet); //returns FALSE
```

Ver também: DEDUP, Filtros do registro

EXP

EXP(n)

n	O número real que será avaliado.
Return:	EXP retorna um valor real único.

A função **EXP** retorna valor exponencial natural do parâmetro (en). Este é o oposto da função LN da .

Exemplo:

```
MyPI := EXP(3.14159);  
Interim := ROUND(1000 * (EXP(MyPI)/(1 + EXP(MyPI))));
```

Ver também: LN, SQRT, POWER

FAIL

`[attrname :=] FAIL [(errormessage | errorcode)] ;`

`[attrname :=] FAIL(errorcode , errormessage) ;`

`[attrname :=] FAIL(datatype [, [errorcode] [, errormessage]]) ;`

<i>attrname</i>	Opcional. O nome da ação, que transforma a ação em definição de atributo, consequentemente não é executado até que <i>attrname</i> seja usado como uma ação.
<i>errormessage</i>	Opcional. Uma constante da string que contém a mensagem a ser exibida.
<i>errorcode</i>	Opcional. Uma constante de número inteiro que contém o número do erro a ser exibido.
<i>datatype</i>	O tipo de valor, nome de uma estrutura RECORD , DATASET, ou DICTIONARY que será emulado.

A ação **FAIL** interrompe imediatamente o processamento na workunit e exibe o *errorcode* e/ou a *errormessage*. A terceira forma está disponível para ser usada em contextos onde é exigido um tipo de valor ou de dataset. FAIL não deve ser usada em um contexto de expressão (tal como no TRANSFORM) – use a ERROR para essas situações.

Exemplo:

```
IF(header.version <> doxie.header_version_new,  
  FAIL('Mismatch -- header.version vs. doxie.header_version_new.'));  
  
FailedJob := FAIL('ouch, it broke');  
sPeople   := SORT(Person,Person.per_first_name);  
nUniques  := COUNT(DEDUP(sPeople,Person.per_first_name AND  
                        Person.address))  
           : FAILURE(FailedJob);  
MyRecSet  := IF(EXISTS(Person),Person,  
                FAIL(Person,99,'Person does not exist!!'));
```

Ver também: FAILURE, ERROR

FAILCODE

FAILCODE

A função **FAILCODE** retorna o último código de falha para ser usado no serviço de fluxo de trabalho **FAILURE** ou na estrutura **TRANSFORM** referenciada na opção **ONFAIL** do **SOAPCALL**

Exemplo:

```
SPeople := SORT(Person, Person.per_first_name);
nUniques := COUNT(DEDUP(sPeople, Person.per_first_name AND
                        Person.address))
:FAILURE(Email.simpleSend(SystemsPersonnel,
SystemsPersonel.email, FAILCODE));
```

Ver também: **FAILURE**, **FAILMESSAGE**, **SOAPCALL**

FAILMESSAGE

FAILMESSAGE [(*tag*)]

<i>tag</i>	Uma constante da string que define o nome da tag XML que contém o texto a ser retornado, normalmente informação adicional retornada pelo SOAPCALL. Se omitida, o padrão é “text”.
------------	---

A função **FAILMESSAGE** retorna a última mensagem de falha para ser usada no serviço de fluxo de trabalho FAILURE ou na TRANSFORM referenciada na opção ONFAIL de SOAPCALL.

Exemplo:

```
SPeople := SORT(Person, Person.per_first_name);
nUniques := COUNT(DEDUP(sPeople, Person.per_first_name AND Person.address))
:FAILURE(Email.simpleSend(SystemsPersonnel,
    SystemsPersonnel.email, FAILMESSAGE));
```

Ver também: RECOVERY, FAILCODE, SOAPCALL

FETCH

FETCH(*basedataset*, *index*, *position* [, *transform*] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>basedataset</i>	O atributo DATASET de base a ser processado. Não é permitido filtro.
<i>index</i>	O atributo INDEX que oferece acesso com chave ao <i>basedataset</i> . Normalmente terá uma expressão de filtro.
<i>position</i>	Uma expressão que oferece meios de localizar o registro correto no <i>basedataset</i> (geralmente o campo no índice que contém o valor de fileposition).
<i>transform</i>	A função TRANSFORM a ser acionada para cada registro buscado no <i>basedataset</i> . Se omitida, FETCH retorna um conjunto contendo todos os campos de ambos <i>basedataset</i> e <i>index</i> , com o segundo de qualquer campo de nome em duplicidade removido.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	FETCH retorna um record set.

A função **FETCH** processa através de todos os registros no *index* na ordem especificada pelo *index* buscando cada registro relacionado do *basedataset* e executando a função *transform*.

O *index* normalmente terá uma expressão de filtro para especificar o conjunto exato de registros a ser retornado do *basedataset*. Se a expressão de filtro definir um único registro no *basedataset*, FETCH retornará apenas esse registro específico. Consulte KEYED/WILD para ler a discussão sobre a filtragem de INDEX.

Requerimentos da Função FETCH TRANSFORM

A função *transform* deve adotar pelo menos dois parâmetros: um registro LEFT que deve estar no mesmo formato que o *basedataset*, e um registro RIGHT opcional que deve estar no mesmo formato que o *index*. O segundo parâmetro opcional é útil em instâncias onde o índice contém informações que não estão presentes no recordset.

Exemplo:

```
PtblRec := RECORD
  STRING2 State := Person.per_st;
  STRING20 City := Person.per_full_city;
  STRING25 Lname := Person.per_last_name;
```

```
    STRING15 Fname := Person.per_first_name;
END;

PtblOut := OUTPUT(TABLE( Person,PtblRec),, 'RTTEMP::TestFetch');
Ptbl := DATASET('RTTEMP::TestFetch',
    {PtblRec,UNSIGNED8 __fpos {VIRTUAL(fileposition)}} ,
    FLAT);

Bld := BUILD(Ptbl,
    {state,city,lname,fname,__fpos},
    'RTTEMPkey::TestFetch');

AlphaInStateCity := INDEX(Ptbl,
    {state,city,lname,fname,__fpos},
    'RTTEMPkey::TestFetch');

TYPEOF(Ptbl) copy(Ptbl 1) := TRANSFORM
    SELF := 1;
END;

AlphaPeople := FETCH(Ptbl,
    AlphaInStateCity(state='FL',
        city ='BOCA RATON',
        Lname='WIK',
        Fname='PICHA'),
    RIGHT.__fpos,
    copy(LEFT));

OutFile := OUTPUT(CHOOSEN(AlphaPeople,10));
SEQUENTIAL(PtblOut,Bld,OutFile)

//NOTE the use of a filter on the index file. This is an important
// use of standard filtering technique in conjunction with indexing
// to achieve optimal "random" access into the base record set
```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, BUILDINDEX, INDEX, KEYED/WILD

FROMJSON

FROMJSON(*record*, *jsonstring* [,**ONFAIL**(*transform*)])

<i>record</i>	A estrutura RECORD a ser gerada. Cada campo deve especificar o XPATH para os dados que devem ser contidos na <i>jsonstring</i> . Se omitido, os nomes do campo de caixa baixa serão utilizados.
<i>jsonstring</i>	Uma string que contém o JSON a ser convertido.
ONFAIL	Opcional. Especifica um transform para manusear os erros no JSON.
<i>transform</i>	Uma estrutura TRANSFORM correspondente à estrutura do registro do primeiro parâmetro.
Return:	FROMJSON retorna uma única linha (registro).

A função **FROMJSON** retorna uma única linha (registro) no formato *record* da *stringjson* especificada. Isso pode ser usado onde quer que uma linha única possa ser usada (semelhante à função ROW).

Exemplo:

```
namesRec := RECORD
  UNSIGNED2 EmployeeID{xpath('EmpID')};
  STRING10 Firstname{xpath('FName')};
  STRING10 Lastname{xpath('LName')};
END;
x := '{"FName": "George" , "LName": "Jetson", "EmpID": 42}';
rec := FROMJSON(namesRec,x);
OUTPUT(rec);
```

Exemplo com manuseio do Error e JSON inválido:

```
namesRec := RECORD
  UNSIGNED2 EmployeeID{xpath('EmpID')};
  STRING20 Firstname{xpath('FName')};
  STRING20 Lastname{xpath('LName')};
END;
x := '{"FName": "malformedJSON" "George" , "LName": "Jetson", "EmpID": 42}';

namesRec createFailure() :=
  TRANSFORM
    SELF.FirstName := FAILMESSAGE;
    SELF := [];
  END;
rec := FROMJSON(namesRec,x,ONFAIL(createFailure()));
OUTPUT(rec);
```

Ver também: ROW, TOJSON

FROMUNICODE

FROMUNICODE(*string*, *encoding*)

<i>string</i>	A string UNICODE para tradução.
<i>encoding</i>	A página do código de codificação (suportada pelo ICU da IBM) a ser usada para tradução.
Return:	FROMUNICODE retorna um único valor de DATA.

A função **FROMUNICODE** retorna a *string* traduzida da *codificação* especificada para um valor DATA.

Exemplo:

```
DATA5 x := FROMUNICODE(u'ABCDE','UTF-8'); //results in 4142434445
```

Ver também: TOUNICODE, UNICODEORDER

FROMXML

FROMXML(*record*, *xmlstring* ,[**ONFAIL**(*transform*)])

<i>record</i>	A estrutura RECORD a ser gerada. Cada campo deve especificar o XPATH para os dados que devem ser contidos na <i>xmlstring</i> .
<i>xmlstring</i>	Uma string que contém o XML a ser convertido.
ONFAIL	Opcional. Especifica um transform para manusear os erros no XML.
<i>transform</i>	Uma estrutura TRANSFORM correspondente à estrutura do registro do primeiro parâmetro.
Return:	FROMXML retorna uma única linha (registro).

A função **FROMXML** retorna uma única linha (registro) no formato *record* da *xmlstring* especificada. Isso pode ser usado onde quer que uma linha única possa ser usada (semelhante à função ROW).

Exemplo:

```
namesRec := RECORD
  UNSIGNED2 EmployeeID{xpath( 'EmpID' ) };
  STRING10  Firstname{xpath( 'FName' ) };
  STRING10  Lastname{xpath( 'LName' ) };
END;
x := '<Row><FName>George</FName><LName>Jetson</LName><EmpID>42</EmpID></Row>';
rec := FROMXML( namesRec, x );
OUTPUT( rec );
```

Exemplo com manuseio do Error e XML inválido:

```
namesRec := RECORD
  UNSIGNED2 EmployeeID{xpath( 'EmpID' ) };
  STRING20  Firstname{xpath( 'FName' ) };
  STRING20  Lastname{xpath( 'LName' ) };
END;
x := '<Row><FName>George</FName><LName><unmatchedtag>Jetson</LName><EmpID>42</EmpID></Row>';

namesRec createFailure() :=
  TRANSFORM
    SELF.FirstName := FAILMESSAGE;
    SELF := [];
  END;
rec := FROMXML( namesRec, x, ONFAIL( createFailure() ) );
OUTPUT( rec );
```

Ver também: ROW, TOXML

GETENV

GETENV(*name* [, *default*])

<i>name</i>	Uma constante da string que contém o nome da variável do ambiente.
<i>default</i>	Opcional. Uma constante da string que contém o valor padrão a ser usado se a variável do ambiente não existir.
Return:	GETENV retorna um valor STRING.

A função **GETENV** retorna o valor da variável de ambiente *name* d. Se a variável do ambiente não existir ou não tiver nenhum valor, o valor *padrão* será retornado.

Exemplo:

```
g1 := GETENV('nameTable');  
g2 := GETENV('myPort', '25');  
  
OUTPUT(g1);
```


GLOBAL

GLOBAL(*expression* [, FEW | MANY])

<i>expression</i>	A expressão que será avaliada em um escopo global.
FEW	Opcional. Indica que a expressão resultará em menos de 10.000 registros. Isso permite otimização para gerar um resultado significativamente mais rápido.
MANY	Opcional. Indica que a expressão resultará em vários registros.
Return:	GLOBAL pode retornar valores scalar ou conjuntos de registro.

A função **GLOBAL** avalia a *expressão* em um escopo global de forma semelhante a do serviço de fluxo de trabalho GLOBAL , mas sem a necessidade de definir outro atributo.

Exemplo:

```
IMPORT doxie;
besr := doxie.best_records;
ssnr := doxie.ssn_records;

/**** Individual record defs
recbesr := RECORDOF(besr);
recssnr := RECORDOF(ssnr);

/**** Monster record def
rec := RECORD, MAXLENGTH(doxie.maxlength_report)
    DATASET(recbesr) best_information_children;
    DATASET(recssnr) ssn_children;
END;
nada := DATASET([0], {INTEGER1 a});
rec tra(nada l) := TRANSFORM
    SELF.best_information_children := GLOBAL(besr);
    SELF.ssn_children := GLOBAL(ssnr);
END;
EXPORT central_records := PROJECT(nada, tra(left));
```

Ver também: Serviço de Fluxo de Trabalho GLOBAL

GRAPH

GRAPH(*recordset* , *iterations* , *processor* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros inicial a ser processado.
<i>iterations</i>	O número de vezes que a função <i>processor</i> (processador) será acionada.
<i>processor</i>	O atributo da função que processará a entrada. Esta função pode usar o seguinte como argumentos:
	ROWSETLEFT Especifica o conjunto de datasets de entrada que podem ser indexados para especificar o recordset de qualquer iteração específica – ROWSET(LEFT)[0] indica o <i>recordset</i> de entrada inicial, enquanto ROWSET(LEFT)[1] indica o conjunto de resultado da primeira iteração. Também pode ser usado como o primeiro parâmetro da função RANGE para especificar um conjunto de datasets (permitindo que o gráfico processe de forma eficiente os argumentos N-ary merge/join).
	COUNTER Especifica um parâmetro INTEGER para o número de iteração gráfica.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	GRAPH retorna o resultado do conjunto de registro da última <i>iteração</i> .

A função **GRAPH** é semelhante a função LOOP , porém é executada como se todas as *iterações* de acionamento do *processador* tivessem sido expandidas, removendo quaisquer ramificações que não pudessem ser executadas e depois unidas. O gráfico resultante é tão eficiente quanto se estivesse sido expandido à mão.

Exemplo:

```
namesRec := RECORD
  STRING20 lname;
  STRING10 fname;
  UNSIGNED2 age := 25;
  UNSIGNED2 ctr := 0;
END;
namesTable2 := DATASET([{'Flintstone','Fred',35},
  {'Flintstone','Wilma',33},
  {'Jetson','Georgie',10},
  {'Mr. T','Z-man'}], namesRec);

loopBody(SET OF DATASET(namesRec) ds, UNSIGNED4 c) :=
  PROJECT(ds[c-1], //ds[0]=original input
  TRANSFORM(namesRec,
```

```
SELF.age := LEFT.age+c; //c is graph COUNTER
SELF.ctr := COUNTER;    //PROJECT's COUNTER
SELF := LEFT));

g1 := GRAPH(namesTable2,10,loopBody(ROWSET(LEFT),COUNTER));

OUTPUT(g1);
```

Ver também: LOOP, RANGE

GROUP

GROUP(*recordset* [, *breakcriteria* [, **ALL**]] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros a ser fragmentado.
<i>breakcriteria</i>	Opcional. Uma lista delimitada por vírgula de expressões ou campos principais no conjunto de registros que especifica como fragmentar o recordset. A palavra-chave RECORD pode ser usada para indicar todos os campos no conjunto de registros, e/ou você pode usar a palavra-chave EXCEPT para listar os campos não agrupados na estrutura. Também é possível usar a palavra ROW para indicar que cada registro no conjunto de registros é um grupo individual. Se omitida, o conjunto de registros será desagrupado de qualquer agrupamento prévio.
ALL	Opcional. Indica que o <i>breakcriteria</i> é aplicado sem levar em conta qualquer ordem prévia. Se omitido, GROUP supõe que o conjunto de registros já foi classificado na ordem do <i>breakcriteria</i> .
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	GROUP retorna um conjunto de registros.

A função **GROUP** fragmenta um *recordset* em um conjunto de valores. Isso permite novas agregações e operações (tais como ITERATE, DEDUP, ROLLUP, SORT e outras) dentro dos subconjuntos definidos dos dados – a operação é executada individualmente em cada subconjunto. Isso significa que o código de condição de limite gravado na função TRANSFORM para essas funções que o utilizam será diferente do código de um *recordset* que tenha sido apenas CLASSIFICADO SORTed.

O *recordset* deve ser classificado através dos mesmos elementos que o *breakcriteria* se a opção ALL não tiver sido especificada.

O *recordset* fica “desagrupado” ao ser usado em uma função TABLE, pela função JOIN em algumas circunstâncias (consulte JOIN), por UNGROUP, ou por outra função GROUP com o segundo parâmetro omitido.

Exemplo:

```
MyRec := RECORD
  STRING20 Last;
  STRING20 First;
END;
SortedSet := SORT(Person, Person.last_name); //sort by last name
GroupedSet := GROUP(SortedSet, last_name); //then group them
```

```
SecondSort := SORT(GroupedSet, Person.first_name);  
    //sorts by first name within each last name group  
    // this is a "sort within group"  
  
UnGroupedSet := GROUP(GroupedSet); //ungroup the dataset  
MyTable := TABLE(SecondSort, MyRec); //create table of sorted names
```

Ver também: REGROUP, COMBINE, UNGROUP, EXCEPT

HASH

HASH(*expressionlist*)

<i>expressionlist</i>	Uma lista de valores delimitada por vírgula.
Return:	HASH retorna um único valor.

A função **HASH** retorna um valor hash de 32 bits derivado de todos os valores da *expressionlist*. Espaços no final da string são removidos (ou UNICODE) antes de o valor ser calculado (a conversão para DATA impede isso).

Exemplo:

```
MySet := DISTRIBUTE(Person, HASH(Person.per_ssn));  
//people with the same SSN go to same Data Refinery node
```

Ver também: DISTRIBUTE, HASH32, HASH64, HASHCRC, HASHMD5

HASH32

HASH32(*expressionlist*)

<i>expressionlist</i>	Uma lista de valores delimitada por vírgula.
Return:	HASH32 retorna um único valor.

A função **HASH32** retorna um valor hash de 32 bits derivado de todos os valores da *expressionlist*. Ela usa um algoritmo hashing que é mais rápido e menos provável do que o **HASH** para retornar os mesmos valores a partir de dados diferentes. Espaços no final da string são removidos (ou UNICODE) antes de o valor ser calculado (a conversão para DATA impede isso).

Exemplo:

```
MySet := DISTRIBUTE(Person, HASH32(Person.per_ssn));  
//people with the same SSN go to same Data Refinery node
```

Ver também: **DISTRIBUTE**, **HASH**, **HASH64**, **HASHCRC**, **HASHMD5**

HASH64

HASH64(*expressionlist*)

<i>expressionlist</i>	Uma lista de valores delimitada por vírgula.
Return:	HASH64 retorna um único valor.

A função **HASH64** retorna um valor hash de 64 bits derivado de todos os valores da *expressionlist*. Espaços no final da string são removidos (ou UNICODE) antes de o valor ser calculado (a conversão para DATA impede isso).

Exemplo:

```
OUTPUT(Person, {per_ssn, HASH64(per_ssn)});  
//output SSN and its 64-bit hash value
```

Ver também: DISTRIBUTE, HASH, HASH32, HASHCRC, HASHMD5

HASHCRC

HASHCRC(*expressionlist*)

<i>expressionlist</i>	Uma lista de valores delimitada por vírgula.
Return:	HASHCRC retorna um único valor.

A função **HASHCRC** retorna um valor de CRC (verificação de redundância cíclica) derivado de todos os valores da *expressionlist*.

Exemplo:

```
OUTPUT(Person, {per_ssn, HASHCRC(per_ssn)});  
//output SSN and its CRC hash value
```

Ver também: DISTRIBUTE, HASH, HASH32, HASH64, HASHMD5

HASHMD5

HASHMD5(*expressionlist*)

<i>expressionlist</i>	Uma lista de valores delimitada por vírgula.
Return:	HASHMD5 retorna um único valor DATA16.

A função **HASHMD5** retorna um valor de hash de 128 bits derivado de todos os valores da *expressionlist*, baseado no algoritmo MD5 desenvolvido pelo professor Ronald L. Rivest do MIT. Diferentemente de outras funções hashing, os espaços no final da string NÃO são removidos antes de o valor ser calculado.

Exemplo:

```
OUTPUT(Person, {per_ssn, HASHMD5(per_ssn)});  
//output SSN and its 128-bit hash value
```

Ver também: DISTRIBUTE, HASH, HASH32, HASH64, HASHCRC

HAVING

HAVING(*groupdataset*, *expression* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>groupdataset</i>	O nome de um conjunto de registro GROUPed.
<i>expression</i>	A expressão lógica pela qual os grupos serão filtrados.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	HAVING retornos um conjunto de registro GROUPed.

A função **HAVING** retorna um conjunto de registros GROUPed> que contém apenas os grupos para os quais a expressão é “true” (verdadeira). Isso é semelhante a cláusula **HAVING** no SQL.

Exemplo:

```
MyGroups := GROUP(SORT(Person,lastname),lastname);  
           //group by last name  
Filtered := HAVING(MyGroups,COUNT(ROWS(LEFT)) > 10);  
           //filter out the small groups
```

Ver também: **GROUP**

HTTPCALL

result := **HTTPCALL**(*url*, *httpmethod*, *responsemime*, *outstructure* [, *options*]);

<i>result</i>	Nome da definição do recordset resultante
<i>url</i>	Uma string que contém a URL que hospeda o serviço a ser chamado. Pode conter parâmetros para o serviço.
<i>httpmethod</i>	Uma string que contém o método HTTP a ser invocado. Os métodos válidos são: "GET"
<i>responsemime</i>	Uma string que contém o tipo de resposta MIME a ser usado. Os tipos válidos são: "text/xml"
<i>outstructure</i>	Uma estrutura RECORD que contém as definições do campo de resultado. Para um <i>responsemime</i> com base em XML, deve ser usado XPATH para especificar o caminho exato dos dados.
<i>options</i>	Uma lista delimitada por vírgula das especificações opcionais da lista abaixo.

HTTPCALL corresponde a uma função que aciona o serviço REST.

Opções válidas são:

RETRY (<i>count</i>)	Especifica quantas vezes houve uma nova tentativa de acionamentos, se ocorrerem erros não fatais. Se omitida, o padrão é três (3).
TIMEOUT (<i>period</i>)	Especifica o número tentativas de leitura antes da falha. O <i>period</i> (período) é um número real cuja parte inteira especifica os segundos. Definir para zero (0) indica espera permanente. Se omitido, o padrão é trezentos (300).
TIMELIMIT (<i>period</i>)	Especifica o tempo total permitido para o HTTPCALL . O <i>period</i> (período) é um número real cuja parte inteira especifica os segundos. Se omitido, o padrão é zero (0) – indicando ausência de limite de tempo.
XPATH (<i>xpath</i>)	Especifica o caminho usado para acessar as linhas no resultado. Se omitido, o padrão é: 'serviceResponse/Results/Result/Dataset/Row'.
ONFAIL (<i>transform</i>)	Especifica acionar a função transform, se o serviço falhar em relação a um registro específico, ou a palavra-chave SKIP. A função TRANSFORM deve gerar o mesmo <i>resultype</i> que <i>outstructure</i> e deve usar FAILCODE e/ou FAILMESSAGE para fornecer detalhes da falha.
TRIM	Especifica que todos os espaços à direita são removidos das strings antes do resultado.
HTTPHEADER	Refere-se às informações de cabeçalho a serem especificadas para o serviço.

Exemplo:

```
worldBankSource := RECORD
  STRING name {XPATH('name')}
END;

OutRec1 := RECORD
  DATASET(worldBankSource) Fred{XPATH('/source')};
END;

raw := HTTPCALL('http://api.worldbank.org/sources', 'GET', 'text/xml', OutRec1, );

OUTPUT(raw);
```

```
////Using HTTPHEADER to pass Authorization info
raw2 := HTTPCALL('http://api.worldbank.org/sources', 'GET', 'text/xml',
                OutRec1, HTTPHEADER('Authorization','Basic dXNlcm5hbWU6cGFzc3dvcmQ='));

OUTPUT(raw2);
```

IF

IF(*expression*, *trueresult* [, *falseresult*])

<i>expression</i>	Uma expressão condicional.
<i>trueresult</i>	O resultado a ser retornado quando a expressão é “true” (verdadeira). Isso pode ser qualquer expressão ou ação.
<i>falseresult</i>	O resultado a ser retornado quando a expressão é “false” (falsa). Isso pode ser qualquer expressão ou ação. Pode ser omitido apenas se o resultado for uma ação.
Return:	IF retorna um único valor, conjunto, conjunto de registros ou ação.

A função **IF** avalia a *expressão* (que deve ser uma expressão condicional com um resultado booleano) e retorna *trueresult* ou *falseresult* com base na avaliação da *expressão*. Ambos *trueresult* e *falseresult* ambos *trueresult* e *falseresult* devem ser do mesmo tipo (ou seja, ambos devem ser strings, conjunto de registros e assim por diante). Se *trueresult* e *falseresult* forem strings, o tamanho da string retornada será do tamanho do valor resultante. Se o código subsequente depender do tamanho dos dois ser o mesmo, então pode ser necessária uma conversão de tipo para o tamanho exigido (normalmente para converter uma string vazia para o tamanho adequado, de modo que a indexação subsequente da string não falhe).

Exemplo:

```
MyDate := IF(ValidDate(Trades.trd_dopn),Trades.trd_dopn,0);
// in this example, 0 is the false value and
// Trades.trd_dopn is the True value returned

MyTrades := IF(person.per_sex = 'Male',
    Trades(trd_bal<100),
    Trades(trd_bal>1000));
// return low balance trades for men and high balance
// trades for women

MyAddress := IF(person.gender = 'M',
    cleanAddress182(person.address),
    (STRING182) '');
//cleanAddress182 returns a 182-byte string
// so casting the empty string false result to a
// STRING182 ensures a proper-length string return
```

Ver também: IFF, MAP, EVALUATE, CASE, CHOOSE, SET

IFF

IFF(*expression*, *trueresult* [, *falseresult*])

<i>expression</i>	Uma expressão condicional.
<i>trueresult</i>	O resultado a ser retornado quando a expressão é “true” (verdadeira). Isso pode ser qualquer expressão ou ação.
<i>falseresult</i>	O resultado a ser retornado quando a expressão é “false” (falsa). Isso pode ser qualquer expressão ou ação. Pode ser omitido apenas se o resultado for uma ação.
Return:	IF retorna um único valor, conjunto, conjunto de registros ou ação.

A função **IFF** desempenha a mesma funcionalidade que a IF, mas assegura que uma *expressão* com lógica booleana complexa seja avaliada exatamente como é exibida.

Ver também: IF, MAP, EVALUATE, CASE, CHOOSE, SET

IMPORT

resulttype funcname (parameterlist) := IMPORT(language, function);

<i>resulttype</i>	O tipo do valor de retorno da <i>função</i> .
<i>funcname</i>	O nome de definição ECL da <i>função</i> .
<i>parameterlist</i>	Uma lista separada por vírgulas com os parâmetros a serem passados para <i>function</i> .
<i>language</i>	Especifica o nome da linguagem de programação externa cujo código você deseja incorporar à sua ECL. O módulo de suporte à linguagem – para a linguagem que deseja incorporar – precisa estar instalado em seu diretório de plugins. São fornecidos módulos para linguagens como Java, R, Javascript e Python. Você pode escrever o seu próprio módulo de suporte à linguagem plugável para qualquer linguagem que ainda não conta com suporte usando os módulos fornecidos como exemplos ou pontos de partida.
<i>function</i>	Uma constante da string que contém o nome da função a ser incluída.

A declaração **IMPORT** permite acionar códigos existentes gravados em *linguagem* externa. Pode ser usada para acionar código Java ou Python, mas não é utilizável com código Javascript ou R (para isso, use a estrutura EMBED). O código Java deve ser colocado em um arquivo .java e compilado da forma usual através do compilador javac. Todas as classes de Java usadas devem ser seguras quanto ao thread (linha de execução).

ATENÇÃO: Esse recurso pode gerar corrupção de memória e/ou problemas de segurança. Portanto, recomendamos cautela e uma consideração detalhada. Consulte o Suporte Técnico antes de usar.

Exemplo:

```
IMPORT Python;

INTEGER addthree(INTEGER p) := IMPORT(Python, 'python_mod_name.addThree');

//Java Example setting the classpath
IMPORT java;
STRING jcat(STRING a, STRING b) :=
    IMPORT(java, 'JavaCat.cat:(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;')
    : classpath('/opt/HPCCSystems/classes/');
jcat('I', 'concatenate');
```

Ver também: IMPORT, Estrutura EMBED

INTFORMAT

INTFORMAT(*expression*, *width*, *mode*)

<i>expression</i>	A expressão que especifica o valor inteiro a ser formatado.
<i>width</i>	O tamanho da string na qual o valor será alinhado à direita.
<i>mode</i>	O formato tipo: 0 = preenchimento em branco, 1 = preenchimento com zero.
Return:	INTFORMAT retorna um único valor.

A função **INTFORMAT** retorna o valor da *expressão* formatada como string justificada à direita dos caracteres de *largura* .

Exemplo:

```
val := 123456789;
OUTPUT(INTFORMAT(val,20,1));
//formats as '00000000000123456789'
OUTPUT(INTFORMAT(val,20,0));
//formats as '          123456789'
```

Ver também: REALFORMAT

ISVALID

ISVALID(*field*)

<i>field</i>	O nome de um campo DECIMAL, REAL ou alien TYPE data.
Return:	ISVALID retorna um único valor booleano.

A função **ISVALID** valida que o *campo* possui um valor legal. Se o conteúdo não for válido para o tipo de valor declarado do *campo* (tais como valores hexadecimais maiores do que 9 em um DECIMAL), ISVALID retorna como FALSE; caso contrário, retorna como TRUE.

Exemplo:

```
MyVal := IF(ISVALID(Infile.DecimalField),Infile.DecimalField,0);  
//ISVALID returns TRUE if the value is legal
```

Ver também: Estrutura TYPE, DECIMAL, REAL

ITERATE

ITERATE(*recordset*, *transform* [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento.
<i>transform</i>	A função TRANSFORM a ser acionada para cada registro no <i>recordset</i> .
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

LOCAL Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.

Return: ITERATE retorna um conjunto de registros.

A função **ITERATE** processa através de todos os registros no *recordset* um par de registros por vez, executando a função *transform* em cada par sucessivamente. O primeiro registro no *recordset* é especificado para *transform* como o primeiro registro RIGHT, emparelhado com um registro LEFT cujos campos estão todos em branco ou definidos para zero. Cada registro resultante de *transform* se torna o registro LEFT do próximo par.

Requerimentos da Função TRANSFORM - ITERATE

A função *transform* deve adotar pelo menos dois parâmetros: Registros LEFT e RIGHT que precisam estar no mesmo formato que o *recordset* resultante. Pode ser especificado um terceiro parâmetro opcional: um COUNTER de número inteiro para especificar o número de vezes que *transform* foi acionado para o *recordset* ou para o grupo atual no *recordset* (consulte a função GROUP).

Exemplo:

```
ResType := RECORD
  INTEGER1 Val;
  INTEGER1 Rtot;
END;

Records := DATASET([ {1,0}, {2,0}, {3,0}, {4,0} ], ResType);
/* these are the recs going in:
Val Rtot
1 0
2 0
3 0
4 0 */
```

```
ResType T(ResType L, ResType R) := TRANSFORM
  SELF.Rtot := L.Rtot + R.Val;
  SELF := R;
END;

MySet1 := ITERATE(Records,T(LEFT,RIGHT));

/* these are the recs coming out:
Val Rtot
1    1
2    3
3    6
4   10 */

//The following code outputs a running balance:
Run_bal := RECORD
  Trades.trd_bal;
  INTEGER8 Balance := 0;
END;
TradesBal := TABLE(Trades,Run_Bal);

Run_Bal DoRoll(Run_bal L, Run_bal R) := TRANSFORM
  SELF.Balance := L.Balance + IF(validmoney(R.trd_bal),R.trd_bal,0);
  SELF := R;
END;

MySet2 := ITERATE(TradesBal,DoRoll(LEFT,RIGHT));
```

See Also: Estrutura TRANSFORM, Estrutura RECORD, ROLLUP

JOIN

JOIN(*leftrecset*, *rightrecset*, *joincondition* [, *transform*] [, *jointype*] [, *joinflags*])

JOIN(*setofdatabases*, *joincondition*, *transform*, **SORTED**(*fields*) [, *jointype*])

<i>leftrecset</i>	O conjunto esquerdo dos registros para processamento.
<i>rightrecset</i>	O conjunto direito dos registros para processamento. Esse pode ser um INDEX.
<i>joincondition</i>	Uma expressão que especifica como combinar registros em <i>leftrecset</i> e <i>rightrecset</i> ou <i>setofdatabases</i> (consulte as discussões de Lógica de combinação abaixo). Na expressão, a palavra-chave LEFT é o qualificador de dataset para os campos no <i>leftrecset</i> e a palavra-chave RIGHT é o qualificador do dataset para os campos no <i>rightrecset</i> .
<i>transform</i>	Opcional. A função TRANSFORM para acionar cada par de registros para processamento. Se omitida, JOIN retorna todos os campos de ambos <i>leftrecset</i> e <i>rightrecset</i> com o segundo de qualquer campo de nome em duplicidade removido.
<i>jointype</i>	Opcional. Se omitida, uma operação de junção interna; caso contrário, um dos tipos listados na seção Tipos de JOIN abaixo.
<i>joinflags</i>	Opcional. Qualquer opção (consulte a seção JOIN Options [OPÇÕES DE JOIN] abaixo) para especificar exatamente como a operação JOIN é executada.
<i>setofdatabases</i>	O SET de conjuntos de registro para processamento ([idx1,idx2,idx3]), normalmente INDEXes (ÍNDICES), em que todos precisam ter o mesmo formato.
SORTED	Especifica a ordem de classificação de registros no <i>setofdatabases</i> de entrada, além da ordem de classificação de resultado no conjunto de resultados.
<i>fields</i>	Uma lista delimitada por vírgulas dos campos no <i>setofdatabases</i> , que precisa ser um subconjunto da ordem de classificação de entrada. Todos esses campos precisam ser usados na <i>joincondition</i> , uma vez que definem a ordem na qual os campos passam pela operação STEPPED.
Return:	JOIN retorna um conjunto de registros.

A função **JOIN** produz um conjunto de resultados baseado na interseção de dois ou mais datasets ou índices (conforme determinado por *joincondition*).

JOIN em Dois Datasets

JOIN(*leftrecset*, *rightrecset*, *joincondition* [, *transform*] [, *jointype*] [, *joinflags*])

A primeira forma de JOIN processa todos os pares de registros em *leftrecset* e *rightrecset* e avalia a *condition* para localizar registros correspondentes. Se a *condition* e *jointype* especificarem que o par de registros se qualifica para processamento, a função *transform* é executada, gerando o resultado.

JOIN classifica/distribui dinamicamente o *leftrecset* e *rightrecset* conforme necessário para realizar sua operação com base na *condition* especificada; assim sendo, **recordset resultante não tem garantia de estar na mesma ordem que os recordsets de entrada**. Se JOIN realizar uma classificação dinâmica de seus recordsets de entrada, essa nova ordem de classificação não pode ser usada para além da execução de JOIN. Esse princípio também se aplica a qualquer função GROUPing – os registros são desagrupados automaticamente conforme necessário, exceto nas circunstâncias a seguir:

* Para junções LOOKUP e ALL, o GROUPing e a ordem de classificação de *leftrecset* são preservados

* Para junções KEYED, o GROUPing (mas não a ordem de classificação) do *leftrecset* é preservado.

Lógica de Correspondência - JOIN

O registro que corresponde a *joincondition* é processado internamente em duas partes:

"equality" (hard match)	Toda a lógica "LEFT.field = RIGHT.field" simples que define os registros correspondentes. Para JOINS que usam chaves, todos esses precisam ser campos na chave para se qualificarem para inclusão nessa parte. Se não houver uma parte de "igualdade" na lógica <i>joincondition</i> , um erro "JOIN too complex" (operação JOIN muito complexa) será exibido.
"non-equality" (soft match)	Todos os outros critérios de correspondência na lógica <i>joincondition</i> , como expressões "LEFT.field > RIGHT.field" ou qualquer lógica OR, que possam estar envolvidos na determinação final de que qualquer registro <i>leftrecset</i> e <i>rightrecset</i> realmente coincidem.

Essa divisão de lógica interna permite que o código JOIN seja otimizado para máxima eficiência – primeiro a lógica de "igualdade" é avaliada para proporcionar um resultado temporário que é então avaliado em relação a qualquer "desigualdade" na *joincondition* correspondente.

Opções

As opções de *joinflags* a seguir podem ser especificadas para determinar exatamente como JOIN é executado.

[, PARTITION LEFT | PARTITION RIGHT | [MANY] LOOKUP [FEW] | GROUPED | ALL | NOSORT [(which)] | KEYED [(index) [, UNORDERED]] | LOCAL | HASH]
[, KEEP(n)] [, ATMOST([condition,] n)] [, LIMIT(value [, SKIP | transform | FAIL])] [, SKEW(limit [, target])] [, THRESHOLD(size)] [, SMART] [, UNORDERED | ORDERED(bool)] [, STABLE | UNSTABLE]
[, PARALLEL [(numthreads)]] [, ALGORITHM(name)]

PARTITION LEFT RIGHT	Especifica qual recordset proporciona os pontos de partição que determinam como os registros são classificados e distribuídos entre os nós do supercomputador. PARTITION RIGHT especifica o <i>rightrecset</i> , enquanto PARTITION LEFT especifica o <i>leftrecset</i> . Se omitido, PARTITION LEFT é o padrão.
[MANY] LOOKUP	Especifica o <i>rightrecset</i> , sendo um arquivo relativamente pequeno de registros de consulta que podem ser copiados completamente para cada nó. MANY não estiver presente, os registros <i>rightrecset</i> têm uma relação de muitos para 0/1 com os registros em <i>leftrecset</i> (para cada registro em <i>leftrecset</i> há no máximo 1 registro em <i>rightrecset</i>). Se MANY estiver presente, os registros <i>rightrecset</i> têm uma relação de muitos para 0/muitos com os registros em <i>leftrecset</i> . Essa opção permite que o otimizador evite a classificação desnecessária do <i>leftrecset</i> . Válido apenas para os <i>jointypes</i> internos, LEFT OUTER ou LEFT ONLY. As opções ATMOST, LIMIT e KEEP são suportadas em conjunto com MANYLOOKUP.
SMART	Especifica para usar uma consulta na memória quando possível, mas usa uma operação de junção distribuída se o dataset direito for muito grande.
FEW	Especifica que o LOOKUP <i>rightrecset</i> possui poucos registros, de forma que pouca memória é utilizada, permitindo que várias junções de consultas sejam incluídas no mesmo subgráfico Thor.
GROUPED	Especifica a mesma ação de MANY LOOKUP, mas preserva o agrupamento Usado principalmente no motor de entrega rápida de dados. Válido apenas para os <i>jointypes</i> internos, LEFT OUTER ou LEFT ONLY. As opções ATMOST, LIMIT e KEEP são suportadas em conjunto com GROUPED.
ALL	Especifica que o <i>rightrecset</i> é um arquivo pequeno que pode ser copiado completamente para cada nó, o que permite que o compilador ignore a falta de qualquer parte de "igualdade" em relação à condição; isso elimina o erro "JOIN too complex" ("operação JOIN muito complexa") que a condição normalmente produziria. Se uma parte de "igualdade" estiver presente, o JOIN

Referência a Linguagem ECL
Ações e Funções Built-in

	é executado internamente como MANY LOOKUP. A opção KEEP é suportada juntamente com essa opção.
NOSORT	Realiza o JOIN sem classificar dinamicamente as tabelas. Isso implica que <i>leftrecset</i> e/ou <i>rightrecset</i> precisam ter sido previamente classificados e particionados com base nos campos especificados em <i>joincondition</i> para que os registros possam ser facilmente combinados.
<i>which</i>	Opcional. As palavras-chave LEFT ou RIGHT para indicar que <i>leftrecset</i> ou <i>rightrecset</i> foram previamente classificados. Se omitidas, NOSORT assume que ambos <i>leftrecset</i> e <i>rightrecset</i> foram classificados previamente.
KEYED	Especifica o uso de acesso indexado em <i>rightrecset</i> (consulte INDEX).
<i>index</i>	Opcional. O nome de um INDEX no <i>rightrecset</i> para um JOIN full-keyed (consulte abaixo). Se omitido, indica que o <i>rightrecset</i> sempre será um INDEX (útil quando <i>rightrecset</i> é especificado como um parâmetro para uma função).
UNORDERED	Opcional. Especifica que a operação KEYED JOIN não preserva a ordem de classificação do <i>leftrecset</i> .
LOCAL	Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
HASH	Especifica um DISTRIBUTE implícito do <i>leftrecset</i> e <i>rightrecset</i> entre os nós de supercomputador com base na <i>joincondition</i> para que cada nó possa realizar seu trabalho com os dados locais.
KEEP(n)	Especifica o número máximo de registros correspondentes(n) a serem gerados no conjunto de resultados de cada registro <i>leftrecset</i> . Se omitido, todas as correspondências são mantidas. Isso é útil onde pode haver muitos pares correspondentes e você precisa limitar o número no conjunto de resultados. KEEP não é suportado para <i>jointypes</i> RIGHT OUTER, RIGHT ONLY, LEFT ONLY ou FULL ONLY.
ATMOST	Especifica o número máximo de registros correspondentes que, se excedido, elimina todas essas correspondências do conjunto de resultados. Isso é útil para situações nas quais você elimina pares de registros com "excesso de correspondências" do conjunto de resultados. ATMOST não é suportado para RIGHT EXTER ONLY ou FULL ONLY <i>jointypes</i> . Há duas formas: ATMOST(condição, n) – o máximo é computado apenas para a condição. ATMOST(n) – o máximo é computado para toda a <i>joincondition</i> , exceto caso KEYED seja usado na <i>joincondition</i> (caso no qual apenas as expressões KEYED são usadas). Quando ATMOST é especificado (e o JOIN não é full-keyed ou half-keyed), a <i>joincondition</i> e a condition podem incluir comparações de campo da string que usam indexação de strings com um asterisco como o limite máximo, como neste exemplo: J1 := JOIN(dsL, dsR, LEFT.name[1..*]=RIGHT.name[3..*] AND LEFT.val < RIGHT.val, T(LEFT,RIGHT), ATMOST(LEFT.name[1..*]=RIGHT.name[3..*],3)); TO asterisco indica correspondência do máximo de caracteres necessários para reduzir o número de possíveis correspondências para abaixo do número ATMOST (n).
<i>condition</i>	Uma parte da expressão <i>joincondition</i> .
<i>n</i>	Especifica o número máximo de correspondências permitidas.
LIMIT	Especifica o número máximo de registros correspondentes que, se excedido, ocasiona a falha da workunit ou elimina todas essas correspondências do conjunto de resultados. Isso é útil para situações nas quais você elimina pares de registros com "excesso de correspondências" do conjunto de resultados. Normalmente usado para JOINS KEYED e "Half-Keyed" (consulte abaixo), LIMIT se difere de ATMOST principalmente por seu efeito em uma junção LEFT OUTER, na qual um registro <i>leftrecset</i> com um número excessivo de registros correspondentes seria tratado como não correspondente pelo ATMOST (o registro <i>leftrecset</i> estaria no resultado sem registros <i>rightrecset</i> correspondentes), enquanto LIMIT faria com que toda

Referência a Linguagem ECL

Ações e Funções Built-in

	a workunit falhasse ou executaria o SKIP no registro (eliminando o registro <i>leftrecset</i> totalmente do resultado). Se omitido, o padrão é LIMIT(10000). O LIMIT é aplicado ao record set que atende à parte de correspondência rígida ("equality") da <i>joincondition</i> , porém antes de a correspondência suave ("desigualdade") da <i>joincondition</i> ser avaliada.
<i>value</i>	O número máximo de correspondências permitido; LIMIT(0) é ilimitado.
SKIP	Opcional. Especifica a eliminação de registros correspondentes que ultrapassam o valor máximo do resultado LIMIT, em vez da falha da workunit.
<i>transform</i>	Opcional. Especifica a emissão de um registro único produzido por <i>transform</i> em vez da falha da workunit (similar à opção ONFAIL da função LIMIT).
FAIL	Opcional. Especifica o uso do FAIL ação para configurar a mensagem de erro quando a workunit falha.
SKEW	Indica que você sabe que os dados para essa junção não serão espalhados uniformemente entre os nós (serão distorcidos após ambos os arquivos terem sido distribuídos baseados na <i>condition</i> da junção), e que você opta por substituir o padrão especificando seu próprio valor limite para permitir que a workunit continue apesar da distorção. Só é válido em junções sem chave (a opção KEYED não está presente e o <i>rightrecset</i> não é um INDEX).
<i>limit</i>	Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima da distorção a ser permitida antes que a workunit falhe (o padrão é 0,1 = 10%).
<i>target</i>	Opcional. Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima desejada da distorção a ser permitida (o padrão é 0,1 = 10%).
THRESHOLD	Indica o tamanho mínimo de uma parte única do <i>leftrecset</i> ou <i>rightrecset</i> antes que o limite de SKEW seja determinado. Só é válido em junções sem chave (a opção KEYED não está presente e o <i>rightrecset</i> não é um INDEX).
<i>size</i>	Um valor inteiro indicando o número mínimo de bytes para uma parte única.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for False, especifica que a ordem do registro de resultado não é importante. Quando for True, especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os threads <i>numthreads</i> .
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

As opções a seguir são mutualmente exclusivas e só podem ser usadas para a exclusão das outras nessa lista. PARTITION LEFT | PARTITION RIGHT | [MANY] LOOKUP | GROUPED | ALL | NOSORT | HASH

Além dessa lista, as opções KEYED e LOCAL também são mutualmente exclusivas com as opções acima listadas, mas não umas com as outras. Quando ambas as opções KEYED e LOCAL são especificadas, apenas as partes INDEX em cada nó são acessadas por esse nó.

Normalmente, o *leftrecset* deve ser maior que o *rightrecset* para evitar problemas de distorção (porque PARTITION LEFT é o comportamento padrão). Se as opções LOOKUP ou ALL forem especificadas, o *rightrecset* precisa ser pequeno o bastante para ser carregado na memória em cada nó e a operação é então implicitamente LOCAL. A opção ALL não é prática se o *rightrecset* for maior que alguns poucos milhares de registros (devido ao número de compara-

ções necessário). O tamanho do *rightrecset* é irrelevante no caso de JOINS "half-keyed" e "full-keyed" (consulte a discussão sobre Join Chaveado).

Use o SMART quando o dataset do lado direito provavelmente for pequeno o bastante para caber na memória, mas sem garantias de tal.

Se aparecer um erro similar a esse:

```
"error: 1301: Pool memory exhausted:..."
```

significa que o *rightrecset* é muito grande e uma operação LOOKUP JOIN não deve ser usada. Uma operação SMART JOIN pode ser uma boa opção neste caso.

JOINS KEYEDs

Uma JOIN full-keyed usa a opção KEYED, e a *joincondition* precisa ser baseada nos campos *index*. Na verdade, a junção é feita entre o *leftrecset* e o *index* no *rightrecset* – o *index* precisa do campo de ponteiro de registro do dataset (VIRTUAL(fileposition)) para buscar corretamente os registros do *rightrecset*. A operação join KEYED típica especifica apenas o *rightrecset* para o TRANSFORM.

Se a *rightrecset* for um INDEX, a operação é uma JOIN "half-keyed". Normalmente, o INDEX em uma JOIN de "half-keyed" contém campos de "payload", que frequentemente eliminam a necessidade de leitura do dataset de base. Se esse for o caso, o INDEX de "payload" não precisa do campo de ponteiro de registro (VIRTUAL(fileposition)) do dataset declarado. Para uma JOIN de half-keyed, a *joincondition* pode usar apenas as palavras-chave KEYED e WILD que estão disponíveis para uso apenas nos filtros INDEX.

Para ambos os tipos de junção KEYED (com chave), qualquer GROUPing dos conjuntos de registro de base permanece inalterado. Consulte KEYED e WILD para ler uma discussão sobre a filtragem de INDEX.

Lógica do Join

A operação JOIN segue a seguinte lógica:

1. Classificação/distribuição de registro para obter possíveis correspondências nos mesmos nós.

As opções PARTITION LEFT, PARTITION RIGHT, LOOKUP, ALL, NOSORT, KEYED, HASH e LOCAL indicam como isso acontece. Essas opções são mutuamente exclusivas; apenas uma pode ser especificada, sendo PARTITION LEFT o padrão. SKEW e THRESHOLD podem modificar o comportamento solicitado. LOOKUP também possui o efeito de deduplicação de *rightrecset* pela *joincondition*.

2. Correspondência de registros.

A *joincondition*, LIMIT e ATMOST determinam como isso é feito.

Um limite implícito de 10000 é adicionado quando não há LIMIT especificado E o seguinte é verdadeiro:

Não há limite especificado para ATMOST E e não é um LEFT ONLY JOIN E (ou nenhum limite KEEP especificado OU o JOIN que tem um postfilter).

3. Determina quais correspondências devem ser especificadas para transform.

O *jointype* determina isso.

4. Gera registros de resultados através da função TRANSFORM.

O parâmetro *transform* implícito ou explícito determina isso.

5. Filtrar registros de resultados com SKIP.

Se *transform* para um par de registros resultar em SKIP, então o registro de resultado não é contado em nenhum total da opção KEEP.

6. Limitar registros de resultados com KEEP.

Quaisquer registros de resultados para um determinado registro *leftrecset* acima e além do valor KEEP permitido são descartados. Em uma junção FULL OUTER, registros *rightrecset* que não correspondem a nenhum registro são tratados como se todos fossem correspondências de diferentes registros *leftrecset* padrão (isto é, o contador KEEP é reiniciado para cada um).

Requerimentos da Função TRANSFORM - JOIN

A função *transform*: um registro LEFT formatado como *leftrecset* e/ou um registro RIGHT formatado como o *rightrecset* (que pode estar em diferentes formatos). O formato do conjunto de registros resultante não precisa ser o mesmo que nenhum conjunto de registros de entrada.

Tipos de Join: Dois DATASETS

Os *jointypes* a seguir produzem os seguintes tipos de resultados com base na correspondência de registros produzida por *joincondition*:

inner (default)	Apenas os registros existentes em ambos <i>leftrecset</i> e <i>rightrecset</i> .
LEFT OUTER	No mínimo um registro para cada um no <i>leftrecset</i> .
RIGHT OUTER	No mínimo um registro para cada um no <i>rightrecset</i> .
FULL OUTER	No mínimo um registro para cada um no <i>leftrecset</i> e <i>rightrecset</i> .
LEFT ONLY	Um registro para cada registro <i>leftrecset</i> sem correspondência no <i>rightrecset</i> .
RIGHT ONLY	Um registro para cada registro <i>rightrecset</i> sem correspondência no <i>leftrecset</i> .
FULL ONLY	Um registro para cada registro <i>leftrecset</i> e <i>rightrecset</i> sem correspondência no conjunto de registro oposto.

Exemplo:

```
outrec := RECORD
  people.id;
  people.firstname;
  people.lastname;
END;

RT_folk := JOIN(people(firstname[1] = 'R'),
               people(lastname[1] = 'T'),
               LEFT.id=RIGHT.id,
               TRANSFORM(outrec,SELF := LEFT));
OUTPUT(RT_folk);

//***** Half KEYED JOIN example:
peopleRecord := RECORD
  INTEGER8 id;
  STRING20 addr;
END;
peopleDataset := DATASET([ {3000, 'LONDON'}, {3500, 'SMITH'},
                           {30, 'TAYLOR'} ], peopleRecord);
PtblRec doHalfJoin(peopleRecord 1) := TRANSFORM
```

```
    SELF := 1;
END;
FilledRecs3 := JOIN(peopleDataset, SequenceKey,
                    LEFT.id=RIGHT.sequence,doHalfJoin(LEFT));
FilledRecs4 := JOIN(peopleDataset, AlphaKey,
                    LEFT.addr=RIGHT.Lname,doHalfJoin(LEFT));

//***** Full KEYED JOIN example:
PtblRec := RECORD
    INTEGER8 seq;
    STRING2  State;
    STRING20 City;
    STRING25 Lname;
    STRING15 Fname;
END;
PtblRec Xform(person L, INTEGER C) := TRANSFORM
    SELF.seq      := C;
    SELF.State    := L.per_st;
    SELF.City     := L.per_full_city;
    SELF.Lname    := L.per_last_name;
    SELF.Fname    := L.per_first_name;
END;
Proj := PROJECT(Person(per_last_name[1]=per_first_name[1]),
                Xform(LEFT,COUNTER));
PtblOut := OUTPUT(Proj, '~RTTEMP::TestKeyedJoin', OVERWRITE);

Ptbl := DATASET('RTTEMP::TestKeyedJoin',
                {PtblRec, UNSIGNED8 __fpos {VIRTUAL(fileposition)}},
                FLAT);
AlphaKey := INDEX(Ptbl, {lname, fname, __fpos},
                  '~RTTEMPkey::lname.fname');
SeqKey := INDEX(Ptbl, {seq, __fpos}, '~RTTEMPkey::sequence');

Bld1 := BUILD(AlphaKey, OVERWRITE);
Bld2 := BUILD(SeqKey, OVERWRITE);
peopleRecord := RECORD
    INTEGER8 id;
    STRING20 addr;
END;
peopleDataset := DATASET([ {3000, 'LONDON'}, {3500, 'SMITH'},
                           {30, 'TAYLOR'} ], peopleRecord);
joinedRecord := RECORD
    PtblRec;
    peopleRecord;
END;
joinedRecord doJoin(peopleRecord l, Ptbl r) := TRANSFORM
    SELF := l;
    SELF := r;
END;

FilledRecs1 := JOIN(peopleDataset, Ptbl, LEFT.id=RIGHT.seq,
                    doJoin(LEFT, RIGHT), KEYED(SeqKey));
FilledRecs2 := JOIN(peopleDataset, Ptbl, LEFT.addr=RIGHT.Lname,
                    doJoin(LEFT, RIGHT), KEYED(AlphaKey));
SEQUENTIAL(PtblOut, Bld1, Bld2, OUTPUT(FilledRecs1), OUTPUT(FilledRecs2))
```

JOIN com Set of Datasets

JOIN(*setofdatabases*, *joincondition*, *transform*, **SORTED**(*fields*) [, *jointype*] [, **UNORDERED** | **ORDERED**(*bool*)]
[, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

A segunda forma de JOIN é similar à função MERGEJOIN no sentido de que usa um SET OF DATASETS como o seu primeiro parâmetro. Isso oferece a possibilidade de unir mais de dois datasets em uma única operação.

Lógica de Correspondência de Registros

O registro correspondente a *joincondition* pode conter duas partes: uma condição STEPPED que pode opcionalmente ser ANDed com condições sem STEPPED. A expressão STEPPED contém as principais expressões de igualdade dos *campos* da opção SORTED (componentes à direita podem ser comparações de intervalo se os valores de intervalo não dependem das linhas LEFT e RIGHT), juntamente com ANDed, usando LEFT e RIGHT como qualificadores de dataset. Caso não esteja presente, a condição STEPPED é deduzida dos *campos* especificados pela opção SORTED.

A ordem dos datasets em *setofdatasets* pode ser importante para a maneira pela qual *joincondition* é avaliado. A *joincondition* é duplicada entre pares adjacentes de datasets, o que significa que essa *joincondition*:

```
LEFT.field = RIGHT.field
```

quando aplicada em um *setofdatasets* de três datasets, é logicamente equivalente a:

```
ds1.field = ds2.field AND ds2.field = ds3.field
```

Requisitos de função do TRANSFORM - JOIN setof-datasets

A função *transform* precisa usar no mínimo um parâmetro que precisa ter uma das duas formas:

LEFT	Formatado como qualquer um dos <i>setofdatasets</i> . Isso indica o primeiro dataset no <i>setofdatasets</i> .
ROWS(LEFT)	Formatado como qualquer um dos <i>setofdatasets</i> . Isso indica um conjunto de registro composto de todos os registros de qualquer dataset no <i>setofdatasets</i> que corresponde ao <i>joincondition</i> – isso não pode incluir todos os datasets no <i>setofdatasets</i> , dependendo de qual <i>jointype</i> for especificado.

O formato do conjunto final de registros de resultado precisa ser o mesmo que os datasets de entrada.

Tipos de Join: setofdatasets

Os *jointypes* a seguir produzem os seguintes tipos de resultados com base na correspondência de registros produzida por *joincondition*:

INNER	Esse é o padrão se nenhum <i>jointype</i> for especificado. Apenas os registros presentes em todos os datasets no <i>setofdatasets</i> .
LEFT OUTER	No mínimo um registro para cada registro no primeiro dataset no <i>setofdatasets</i> .
LEFT ONLY	Um registro para cada um no primeiro dataset no <i>setofdatasets</i> para o qual não há correspondência em nenhum dos datasets subsequentes.
MOFN(min [,max])	Um registro para cada registro com registros correspondentes no número mínimo de datasets adjacentes no <i>setofdatasets</i> . Se o número máximo for especificado, o registro não será incluído se o número máximo de correspondências de datasets for excedido.

Exemplo:

```
Rec := RECORD,MAXLENGTH(4096)
  STRING1 Letter;
  UNSIGNED1 DS;
  UNSIGNED1 Matches := 0;
  UNSIGNED1 LastMatch := 0;
  SET OF UNSIGNED1 MatchDSs := [];
END;
```

```
ds1 := DATASET([{'A',1},{ 'B',1},{ 'C',1},{ 'D',1},{ 'E',1}],Rec);
ds2 := DATASET([{'A',2},{ 'B',2},{ 'H',2},{ 'I',2},{ 'J',2}],Rec);
ds3 := DATASET([{'B',3},{ 'C',3},{ 'M',3},{ 'N',3},{ 'O',3}],Rec);
ds4 := DATASET([{'A',4},{ 'B',4},{ 'R',4},{ 'S',4},{ 'T',4}],Rec);
ds5 := DATASET([{'B',5},{ 'V',5},{ 'W',5},{ 'X',5},{ 'Y',5}],Rec);
SetDS := [ds1,ds2,ds3,ds4,ds5];

Rec XF(Rec L,DATASET(Rec) Matches) := TRANSFORM
  SELF.Matches      := COUNT(Matches);
  SELF.LastMatch    := MAX(Matches,DS);
  SELF.MatchDSs     := SET(Matches,DS);
  SELF := L;
END;

j1 := JOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  XF(LEFT,ROWS(LEFT)),SORTED(Letter));
j2 := JOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  XF(LEFT,ROWS(LEFT)),SORTED(Letter),LEFT OUTER);
j3 := JOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  XF(LEFT,ROWS(LEFT)),SORTED(Letter),LEFT ONLY);
j4 := JOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  XF(LEFT,ROWS(LEFT)),SORTED(Letter),MOFN(3));
j5 := JOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  XF(LEFT,ROWS(LEFT)),SORTED(Letter),MOFN(3,4));

OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(j4);
OUTPUT(j5);
```

Ver também: Estrutura TRANSFORM,Estrutura RECORD,SKIP ,STEPPED, KEYED/WILD, MERGEJOIN

KEYDIFF

[*attrname* :=] **KEYDIFF**(*index1*, *index2*, *file* [, **OVERWRITE**] [, **EXPIRE**([*days*])] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

<i>attrname</i>	Opcional. O nome da ação transforma a ação em uma definição de atributo; consequentemente, não é executado até que <i>attrname</i> seja usado como ação.
<i>index1</i>	Um atributo INDEX
<i>índice</i>	Um atributo INDEX cuja estrutura é idêntica ao <i>index1</i> .
<i>file</i>	Uma constante da string que especifica o nome lógico do arquivo para o qual as diferenças serão gravadas.
OVERWRITE	Opcional. Especifica a substituição do nome do arquivo caso ele exista.
EXPIRE	Opcional. Especifica que se trata de um arquivo temporário que pode ser removido automaticamente após um determinado número de dias.
<i>days</i>	Opcional. O número de dias em que o arquivo será automaticamente removido. Se omitido, o padrão é sete (7).
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os threads <i>numthreads</i> . <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .

A ação **KEYDIFF** compara o *index1* com o *index2* e grava as diferenças no *arquivo* especificado. Se a comparação do *index1* com o *index2* não apresentar exatamente a mesma estrutura, ocorrerá um erro. Após ter sido gerado, o *arquivo* pode ser usado pela ação **KEYPATCH**.

Exemplo:

```
Vehicles := DATASET('vehicles',
  {STRING2 st,
   STRING20 city,
   STRING20 lname,
   UNSIGNED8 filepos{VIRTUAL(fileposition)}}},
  FLAT);

i1 := INDEX(Vehicles,
  {st,city,lname,filepos},
  'vkey::20041201::st.city.lname');
i2 := INDEX(Vehicles,
  {st,city,lname,filepos},
  'vkey::20050101::st.city.lname');

KEYDIFF(i1,i2,'KEY::DIFF::20050101::i1i2',OVERWRITE);
```

Ver também: **KEYPATCH**, **INDEX**

KEYPATCH

[*attrname* :=] **KEYPATCH**(*index*, *patchfile*, *newfile* [, **OVERWRITE**] [, **EXPIRE**([*days*])] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

<i>attrname</i>	Opcional. O nome da ação transforma a ação em uma definição de atributo; consequentemente, não é executado até que <i>attrname</i> seja usado como ação.
<i>index</i>	O atributo INDEX para o qual as mudanças serão aplicadas.
<i>patchfile</i>	Uma constante da string que especifica o nome lógico do arquivo que contém as mudanças a serem implementadas (criado por KEYDIFF).
<i>newfile</i>	Uma constante da string que especifica o nome lógico do arquivo para o qual o novo índice será gravado.
OVERWRITE	Opcional. Especifica a substituição do <i>newfile</i> caso ele exista.
EXPIRE	Opcional. Especifica que o <i>newfile</i> é um arquivo temporário que pode ser removido automaticamente após um número especificado de dias.
<i>days</i>	Opcional. O número de dias em que o arquivo será automaticamente removido. Se omitido, o padrão é sete (7).
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

A ação **KEYPATCH** usa o *index* e *patchfile* para gravar um novo índice no *newfile* especificado que contém todos os dados originais do índice atualizados pelas informações do *patchfile*.

Exemplo:

```
Vehicles := DATASET('vehicles',
  {STRING2 st,
   STRING20 city,
   STRING20 lname,
   UNSIGNED8 filepos{VIRTUAL(fileposition)}},
  FLAT);
i1 := INDEX(Vehicles,
  {st,city,lname,filepos},
  'vkey::20041201::st.city.lname');
i2 := INDEX(Vehicles,
  {st,city,lname,filepos},
  'vkey::20050101::st.city.lname');
a := KEYDIFF(i1,i2,'KEY::DIFF::20050101::i1i2',OVERWRITE);
b := KEYPATCH(i1,
  'KEY::DIFF::20050101::i1i2',
  'vkey::st.city.lname'OVERWRITE);
```

`SEQUENTIAL(a,b);`

Ver também: KEYDIFF, INDEX

KEYUNICODE

KEYUNICODE(*string*)

<i>string</i>	Uma string UNICODE.
Return:	KEYUNICODE retorna um único valor de DATA.

A função **KEYUNICODE** retorna um valor de DADOS DATA derivado do parâmetro da *string* de modo que uma comparação destes valores de dados é equivalente a uma comparação sensível à localidade dos valores Unicode que os geraram – e, por ser um simples memcmp(), é significativamente mais rápido. Os valores da *string* que estão sendo gerados precisam ser da mesma localidade ou os resultados serão imprevisíveis. Esta função é especialmente útil se você estiver realizando várias comparações em um campo UNICODE de um dataset grande – sugerimos gerar um campo de chave e realizar as comparações neste campo.

Exemplo:

```
//where you might do this:
my_record := RECORD
  UNICODE_en_US str;
END;
my_dataset := DATASET('filename', my_record, FLAT);
my_sorted := SORT(my_dataset, str);
//you could instead do this:
my_record := RECORD
  UNICODE_en_US str;
  DATA strkey := KEYUNICODE(SELF.str);
END;
my_dataset := DATASET('filename', my_record, FLAT);
my_sorted := SORT(my_dataset, strkey);
```

Ver também: UNICODE, LOCALE

LENGTH

LENGTH(*expression*)

<i>expression</i>	Uma expressão sting.
Return:	LENGTH retorna um único valor inteiro.

A função **LENGTH** retorna o comprimento da string resultante da *expressão* ao tratar a *expressão* como uma STRING temporária.

Exemplo:

```
INTEGER MyLength := LENGTH('XYZ' + 'ABC');  
//MyLength is 6
```

Ver também: Operadores String, STRING

LIBRARY

LIBRARY(**INTERNAL**(*module*), *interface* [(*parameters*)])

LIBRARY(*module* , *interface* [(*parameters*)])

INTERNAL	Opcional. Especifica que o módulo é um atributo, não uma biblioteca externa (criado pela ação BUILD).
<i>module</i>	O nome da biblioteca de consulta. Quando for INTERNAL, este é o nome do atributo MODULE que implementa a biblioteca de consulta. Se não for INTERNAL, é uma expressão da string que contém o nome da tarefa que compilou a biblioteca de consulta (normalmente definido com #WORKUNIT).
<i>interface</i>	O nome da estrutura INTERFACE que define a biblioteca de consulta.
<i>parameters</i>	Opcional. Os valores a serem especificados para a INTERFACE, se configurada para receber parâmetros.
Return:	LIBRARY resulta em um MODULE que pode ser usado para referenciar os atributos exportados de um módulo específico.

A função **LIBRARY** define uma instância de uma biblioteca de consulta – a *interface* implementada pelo *module* após ter passado pelos parâmetros *especificados*. **As bibliotecas de consulta são usadas apenas pelo hthor e Roxie.**

As bibliotecas INTERNAS são normalmente usadas no desenvolvimento de consultas, ao passo que as bibliotecas externas são melhores para gerar consultas. Uma biblioteca interna gera o código de biblioteca como uma unidade separada, mas depois inclui essa unidade na tarefa de consulta. Ela não possui a vantagem de redução do tempo de compilação ou do uso de memória no Roxie (vantagens da biblioteca externa), mas retém a estrutura da biblioteca, o que significa que as alterações no código não podem afetar mais ninguém que esteja usando o sistema.

As bibliotecas externas são criadas pela ação BUILD e usam uma forma de “nome” #WORKUNIT para especificar o nome externo da biblioteca. Uma biblioteca externa é previamente compilada e, por isso, reduz o tempo de compilação das consultas que a utilizam. A biblioteca também diminui o uso da memória no Roxie.

Exemplo:

```
NamesRec := RECORD
    INTEGER1 NameID;
    STRING20 FName;
    STRING20 LName;
END;
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                        NamesRec);
FilterLibIfacel(DATASET(namesRec) ds, STRING search) := INTERFACE
    EXPORT DATASET(namesRec) matches;
    EXPORT DATASET(namesRec) others;
END;
FilterDsLib1(DATASET(namesRec) ds, STRING search) :=
    MODULE,LIBRARY(FilterLibIfacel)
    EXPORT matches := ds(Lname = search);
    EXPORT others := ds(Lname != search);
END;

// Run this to create the 'Ppass.FilterDsLib' external library
// #WORKUNIT('name','Ppass.FilterDsLib')
// BUILD(FilterDsLib);
```

```
lib1 := LIBRARY(INTERNAL(FilterDsLib1),
  FilterLibIface1(NamesTable, 'Holliday'));
lib2 := LIBRARY('Ppass.FilterDsLib',
  FilterLibIface1(NamesTable, 'Holliday'));
IFilterArgs := INTERFACE
  EXPORT DATASET(namesRec) ds;
  EXPORT STRING search;
END;
FilterLibIface2(IFilterArgs args) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;

FilterDsLib2(IFilterArgs args) := MODULE, LIBRARY(FilterLibIface2)
  EXPORT matches := args.ds(Lname = args.search);
  EXPORT others := args.ds(Lname != args.search);
END;
// Run this to create the 'Ipass.FilterDsLib' external library
// #WORKUNIT('name', 'Ipass.FilterDsLib')
// BUILD(FilterDsLib2);
SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;
lib3 := LIBRARY(INTERNAL(FilterDsLib2),
  FilterLibIface2(SearchArgs));
lib4 := LIBRARY('Ipass.FilterDsLib',
  FilterLibIface2(SearchArgs));

OUTPUT(lib1.matches, NAMED('INTERNAL_matches_straight_parms'));
OUTPUT(lib1.others, NAMED('INTERNAL_nonmatches_straight_parms'));
OUTPUT(lib2.matches, NAMED('EXTERNAL_matches_straight_parms'));
OUTPUT(lib2.others, NAMED('EXTERNAL_nonmatches_straight_parms'));
OUTPUT(lib3.matches, NAMED('INTERNAL_matches_interface_parms'));
OUTPUT(lib3.others, NAMED('INTERNAL_nonmatches_interface_parms'));
OUTPUT(lib4.matches, NAMED('EXTERNAL_matches_interface_parms'));
OUTPUT(lib4.others, NAMED('EXTERNAL_nonmatches_interface_parms'));
```

LIMIT

LIMIT(*recset*, *maxrecs* [, *failclause*] [, **KEYED** [, **COUNT**]] [, **SKIP** [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

LIMIT(*recset*, *maxrecs* [, **ONFAIL**(*transform*)] [, **KEYED** [, **COUNT**]] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recset</i>	O conjunto de registros a limitar. Pode ser um INDEX ou qualquer expressão que gere um resultado no conjunto de registros.
<i>maxrecs</i>	O número máximo de registros permitido em um único nó do supercomputador.
<i>failclause</i>	Opcional. Um acionamento de serviço de fluxo trabalho FAIL padrão.
KEYED	Opcional. Especifica a limitação da parte com chave de uma leitura INDEX .
COUNT	Opcional. Especifica que o limite KEYED foi verificado previamente usando uma keyspan.
SKIP	Opcional. Especifica que quando o limite é excedido, ela é simplesmente eliminada de qualquer resultado em vez de falhar a workunit.
ONFAIL	Opcional. Especifica a emissão de um único registro produzido por transform em vez da falha da workunit.
<i>transform</i>	A função TRANSFORM a ser acionada para gerar um único registro de resultado.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

A função **LIMIT** ocasiona a falha do atributo, exceto se o *recset* possuir mais registros do que o *maxrecs* em qualquer nó único do supercomputador (a menos que a opção SKIP seja usada para a leitura de um índice ou se a opção ONFAIL estiver presente). Se *failclause* estiver presente, ela especifica o número e a mensagem de exceção. Isso é normalmente usado para controlar consultas “sem controle” no supercomputador do Motor de entrega rápida de dados.

Exemplo:

```
RecStruct := RECORD
  INTEGER1 Number;
  STRING1 Letter;
END;
SomeFile := DATASET([ {1, 'A'}, {1, 'B'}, {1, 'C'}, {1, 'D'}, {1, 'E'},
                     {1, 'F'}, {1, 'G'}, {1, 'H'}, {1, 'I'}, {1, 'J'},
                     {2, 'K'}, {2, 'L'}, {2, 'M'}, {2, 'N'}, {2, 'O'},
                     {2, 'P'}, {2, 'Q'}, {2, 'R'}, {2, 'S'}, {2, 'T'},
                     {2, 'U'}, {2, 'V'}, {2, 'W'}, {2, 'X'}, {2, 'Y'} ],
                    RecStruct);
//throw an exception
```

```
X := LIMIT(SomeFile,10, FAIL(99,'error!'));  
//single record output  
Y := LIMIT(SomeFile,10,  
    ONFAIL(TRANSFORM(RecStruct,  
        SELF := ROW({0,''},RecStruct))));  
//no exception, just no record  
Z := LIMIT(SomeFile,10,SKIP);
```

Ver também: FAIL, TRANSFORM

LN

LN(*n*)

n	O número real que será avaliado.
Return:	LN retorna um valor real único.

A função **LN** retorna o logaritmo natural do parâmetro. Trata-se do oposto da função EXP

Exemplo:

```
MyLogPI := LN(3.14159); //1.14473
```

Ver também: EXP, SQRT, POWER, LOG

LOADXML

[*attributename* :=] **LOADXML**(*xmlstring* / *symbol* [, *branch*])

<i>attributename</i>	Opcional. O nome da ação, que transforma a ação em definição de atributo, consequentemente não é executado até que <i>attributename</i> seja usado como uma ação.
<i>xmlstring</i>	Uma expressão da string que contém o texto XML a ser processado em linha (sem retornos ou alimentações de linha).
<i>symbol</i>	O símbolo modelo que contém o texto XML a ser processado (normalmente carregado por #EXPORT ou #EXPORTXML).
<i>branch</i>	Uma string definida pelo usuário que nomeia o texto XML , permitindo assim a operação de #FOR.

LOADXML abre um escopo XML ativo para que as declarações de linguagem de modelo ou símbolos possam atuar. **LOADXML** precisa ser a primeira linha do código para funcionar corretamente.

LOADXML também é usado no código MACRO de “detalhamento”.

Exemplo:

```
LOADXML('<section><item type="count"><set>person</set></item></section>')
//this macro receives in-line XML as its parameter
//and demonstrates the code for multiple row drilldown
EXPORT id(xmlRow) := MACRO
STRING myxmlText := xmlRow;
LOADXML(myxmlText);
#DECLARE(OutStr)
#SET(OutStr, '' )
#FOR(row)
    #APPEND(OutStr,
        'OUTPUT(FETCH(Files.People,Files.PeopleIDX(id='
        + %'id'% + '),RIGHT.RecPos));\n' )
    #APPEND(OutStr,
        'ds' + %'id'%
        + ' := FETCH(Files.Property,Files.PropertyIDX(personid= '
        + %'id'% + '),RIGHT.RecPos);\n' )
    #APPEND(OutStr,
        'OUTPUT(ds' + %'id'%
        + ',{countTaxdata := COUNT(Taxrecs), ds'
        + %'id'% + '});\n' )
    #APPEND(OutStr,
        'OUTPUT(FETCH(Files.Vehicle,Files.VehicleIDX(personid= '
        + %'id'% + '),RIGHT.RecPos));\n' )
#END
%OutStr%
ENDMACRO;

//this is an example of code for a drilldown (1 per row)
EXPORT CountTaxdata(xmlRow) := MACRO
LOADXML(xmlRow);
OUTPUT(FETCH(Files.TaxData,
    Files.TaxdataIDX(propertyid=%propertyid%),
    RIGHT.RecPos));
ENDMACRO;

//This example uses #EXPORT to generate the XML
NamesRecord := RECORD
    STRING10 first;
```



```
    STRING20 last;
END;
r := RECORD
    UNSIGNED4 dg_parentid;
    STRING10  dg_firstname;
    STRING    dg_lastname;
    UNSIGNED1 dg_prange;
    IFBLOCK(SELf.dg_prange % 2 = 0)
        STRING20 extrafield;
    END;
    NamesRecord namerec;
    DATASET(NamesRecord) childNames;
END;

ds := DATASET('~RTTEST::OUT::ds', r, thor);

//Walk a record and do some processing on it.
#DECLARE(out)
#EXPORT(out, r);
LOADXML('%out%', 'FileStruct');

#FOR (FileStruct)
    #FOR (Field)
        #IF (%'{@isEnd}'% <> '')
OUTPUT('END');
        #ELSE
OUTPUT('%'{@type}'%
    #IF (%'{@size}'% <> '-15' AND
        '%'{@isRecord}'%='' AND
        '%'{@isDataset}'%='')
+ '%'{@size}'%
        #END
+ ' ' + '%'{@label}'% + ';');
        #END
    #END
#END
OUTPUT('Done');
```

Ver também: [Templates](#), [#EXPORT](#), [#EXPORTXML](#)

LOCAL

LOCAL(*data*)

<i>data</i>	O nome do atributo de DATASET e INDEX
Return:	THISNODE retorna um conjunto de registros ou índice.

A função **LOCAL** especifica que todas as operações subsequentes nos *dados* são executadas localmente em cada nó (semelhante ao uso da opção LOCAL em uma função). Ela é geralmente usada em uma operação ALLNODES . **Essa função está disponível para uso no cluster Roxie.**

Exemplo:

```
ds := JOIN(SomeData,LOCAL(SomeIndex), LEFT.ID = RIGHT.ID);
```

Ver também: ALLNODES, THISNODE, NOLOCAL

LOG

LOG(*n*)

n	O número real que será avaliado.
Return:	LOG retorna um valor real único.

A função **LOG** retorna o logaritmo de base 10 do parâmetro.

Exemplo:

```
MyLogPI := LOG(3.14159); //0.49715
```

Ver também: EXP, SQRT, POWER, LN

LOOP

LOOP(*dataset*, *loopcount*, *loopbody* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)] [, **FEW**])

LOOP(*dataset*, *loopcount*, *loopfilter*, *loopbody* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)] [, **FEW**])

LOOP(*dataset*, *loopfilter*, *loopbody* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)] [, **FEW**])

LOOP(*dataset*, *rowfilter*, *loopcondition*, *loopbody* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)] [, **FEW**])

LOOP(*dataset*, *loopfilter*, *loopcondition*, *loopbody* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)] [, **FEW**])

<i>dataset</i>	O conjunto de registros a ser processado.
<i>loopcount</i>	Uma expressão de número inteiro que especifica o número de vezes a iterar.
<i>loopbody</i>	A operação a ser executada iterativamente. Pode ser PROJECT, JOIN, ou outra operação do tipo. ROWS(LEFT) é sempre usado como o primeiro parâmetro da operação, indicando que o dataset especificado é o parâmetro de entrada.
<i>loopfilter</i>	Uma expressão lógica que especifica o conjunto de registros cujo processamento ainda não foi concluído. O conjunto de registros que não atende à condição não será mais processado iterativamente e será colocado no final do conjunto de resultado. Esta avaliação ocorre antes de cada iteração do <i>loopbody</i> .
<i>loopcondition</i>	Uma expressão lógica que especifica a continuidade da iteração de <i>loopbody</i> enquanto for TRUE. <i>loopbody</i> iteração enquanto TRUE.
<i>rowfilter</i>	Uma expressão lógica que especifica um único registro cujo processamento foi concluído. O registro que atende à condição não será mais processado iterativamente e será colocado no final do conjunto de resultado. Esta avaliação ocorre durante de os iteração do <i>loopbody</i> .
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
FEW	Opcional. Indica que as atividades não exigirão uma grande quantidade de memória. Isso pode reduzir o número de subgráficos gerados em um LOOP, que diminui a sobrecarga. Usar apenas em consultas Thor.
Return:	LOOP retorna um conjunto de registros.

A função **LOOP** executa iterativamente a operação *loopbody*. O está implícito e disponível para uso para retornar a iteração atual.

A opção PARALLEL

A opção PARALLEL permite que múltiplas iterações de loop sejam executadas em paralelo.

Há uma restrição: ROWS(LEFT) não pode ser usado diretamente em uma subconsulta de *loopbody*.

Exemplo:

```
namesRec := RECORD
  STRING20 lname;
  STRING10 fname;
  UNSIGNED2 age := 25;
  UNSIGNED2 ctr := 0;
END;
namesTable2 := DATASET([{'Flintstone','Fred',35},
  {'Flintstone','Wilma',33},
  {'Jetson','Georgie',10},
  {'Mr. T','Z-man'}], namesRec);
loopBody(DATASET(namesRec) ds, unsigned4 c) :=
  PROJECT(ds,
    TRANSFORM(namesRec,
      SELF.age := LEFT.age*c;
      SELF.ctr := COUNTER ;
      SELF := LEFT));
//Form 1:
OUTPUT(LOOP(namesTable2,
  COUNTER <= 10,
    PROJECT(ROWS(LEFT),
      TRANSFORM(namesRec,
        SELF.age := LEFT.age*2;
        SELF.ctr := LEFT.ctr + COUNTER ;
        SELF := LEFT))));
OUTPUT(LOOP(namesTable2, 4, ROWS(LEFT) & ROWS(LEFT)));
//Form 2:
OUTPUT(LOOP(namesTable2,
  10,
  LEFT.age * COUNTER <= 200,
    PROJECT(ROWS(LEFT),
      TRANSFORM(namesRec,
        SELF.age := LEFT.age*2;
        SELF := LEFT))));
//Form 3:
OUTPUT(LOOP(namesTable2,
  LEFT.age < 100,
  loopBody(ROWS(LEFT), COUNTER)));
//Form 4:
OUTPUT(LOOP(namesTable2,
  SUM(ROWS(LEFT), age) < 1000 * COUNTER,
    PROJECT(ROWS(LEFT),
      TRANSFORM(namesRec,
        SELF.age := LEFT.age*2;
        SELF := LEFT))));
//Form 5:
OUTPUT(LOOP(namesTable2,
  LEFT.age < 100,
  EXISTS(ROWS(LEFT)) and SUM(ROWS(LEFT), age) < 1000,
  loopBody(ROWS(LEFT), COUNTER)));
```

MAP

MAP(*expression* => *value*, [*expression* => *value*, ...] [, *elsevalue*])

<i>expression</i>	Uma expressão condicional.
=>	O operador "resulta em" - válido somente em CHOOSESETS, CASE e MAP.
<i>value</i>	O valor a ser retornado se a expressão for “true” (verdadeira). Pode ser uma expressão de um único valor, um conjunto de valores, um DATASET, um DICTIONARY, um recordset ou uma ação.
<i>elsevalue</i>	Opcional. O valor a ser retornado se todas as expressões forem “false” (falso). Pode ser uma expressão de um único valor, um conjunto de valores, um conjunto de registros ou uma ação. Pode ser omitido se todos os valores de retorno forem ações (o padrão então seria nenhuma ação), ou se todos os valores de retorno forem conjuntos de registro (o padrão então seria um conjunto de registros vazio).
Return:	MAP retorna um único <i>valor</i> .

A função **MAP** avalia a lista de *expressões* e retorna o *valor* associado com a primeira *expressão* “true” (verdadeira). Se não houver correspondência, *elsevalue* será retornado. MAP pode ser considerado como um tipo de estrutura “IF ... ELIF ... ELSE”.

Todos os valores de retorno de *value* e *elsevalue* devem ser exatamente do mesmo tipo, ou ocorrerá um erro de “incompatibilidade de tipo”. Todas as *expressões* devem referenciar o mesmo nível de escopo do dataset, ou ocorrerá um erro de “escopo inválido”. Por consequência, todas as *expressões* devem referenciar os campos em um mesmo dataset ou a existência de um conjunto de registros secundários relacionados (consulte EXISTS).

As *expressões* são normalmente avaliadas na ordem em que são exibidas, porém se todos os *valores* de retorno forem escalar, o otimizador de código pode alterar essa ordem.

Exemplo:

```
Attr01 := MAP(EXISTS(Person(Person.EyeColor = 'Blue')) => 1,
              EXISTS(Person(Person.Haircolor = 'Brown')) => 2,
              3);
//If there are any blue-eyed people, Attr01 gets 1
//elsif there are any brown-haired people, Attr01 gets 2
//else, Attr01 gets 3

Valu6012 := MAP(NoTrades => 99,
                 NoValidTrades => 98,
                 NoValidDates => 96,
                 Count6012);
//If there are no trades, Valu6012 gets 99
//elsif there are no valid trades, Valu6012 gets 98
//elsif there are no valid dates, Valu6012 gets 96
//else, Valu6012 gets Count6012

MyTrades := MAP(rms.rms14 >= 93 => trades(trd_bal >= 10000),
                 rms.rms14 >= 2 => trades(trd_bal >= 2000),
                 rms.rms14 >= 1 => trades(trd_bal >= 1000),
                 Trades);
// this example takes the value of rms.rms14 and returns a
// set of trades based on that value. If the value is <= 0,
// then all trades are returned.
```

Ver também: EVALUATE, IF, CASE, CHOOSE, CHOOSESETS, REJECTED, WHICH

MAX

MAX (*recordset*, *value* [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL**] [, (*numthreads*)] [, **ALGORITHM**(*name*)])

MAX(*valuelist*)

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Também pode ser a palavra-chave GROUP para indicar a localização do valor máximo do campo em um grupo, quando usada em uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>value</i>	A expressão da qual o valor máximo será localizado.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
<i>valuelist</i>	Uma lista delimitada por vírgula das expressões das quais o valor máximo será localizado. Também pode ser um SET de valores.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	MAX retorna um único valor.

A função **MAX** retorna o *valor* máximo do *recordset* especificado ou a *valuelist*. Está configurada para retornar o valor zero caso o *recordset* esteja vazio.

Exemplo:

```
MaxVal1 := MAX(Trades, Trades.trd_rate);
MaxVal2 := MAX(4,8,16,2,1); //returns 16
SetVals := [4,8,16,2,1];
MaxVal3 := MAX(SetVals); //returns 16
```

Ver também: MIN, AVE

MERGE

MERGE(*recordsetlist* , **SORTED**(*fieldlist*) [, **DEDUP**] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

MERGE(*recordsetset* , *fieldlist* , **SORTED**(*fieldlist*) [, **DEDUP**] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordsetlist</i>	Uma lista delimitada por vírgula dos datasets e índices a serem fundidos, os quais todos devem estar exatamente no mesmo formato e ordem de classificação.
SORTED	Especifica a ordem de classificação do <i>recordsetlist</i> .
<i>fieldlist</i>	Uma lista delimitada por vírgula dos campos que definem a ordem de classificação.
DEDUP	Opcional. Especifica que os resultados contêm apenas registros com valores únicos nos campos que especificam a <i>fieldlist</i> da ordem de classificação.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
<i>recordsetset</i>	Um CONJUNTO ([ds1,ds2,ds3]) de datasets ou índices a ser fundido, o qual deve estar exatamente no mesmo formato.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	MERGE retorna um conjunto de registros.

A função **MERGE** retorna um único dataset ou índice contendo todos os registros dos datasets ou índices nomeados no *recordsetlist* ou *recordsetset*. Isso é especialmente útil para atualizações gradativas de dados, uma vez que permite a fusão de um conjunto menor de novos registros em um dataset ou índice grande existente sem a necessidade de reprocessar todos os dados de origem. A forma *recordsetset* possibilita a fusão de vários números de datasets quando usada dentro da função **GRAPH**.

Exemplo:

```
ds1 := SORTED(DATASET([ {1,'A'}, {1,'B'}, {1,'C'}, {1,'D'}, {1,'E'},
                        {1,'F'}, {1,'G'}, {1,'H'}, {1,'I'}, {1,'J'} ],
                {INTEGER1 number, STRING1 Letter} ),
            letter, number);
ds2 := SORTED(DATASET([ {2,'A'}, {2,'B'}, {2,'C'}, {2,'D'}, {2,'E'},
                        {2,'F'}, {2,'G'}, {2,'H'}, {2,'I'}, {2,'J'} ],
                {INTEGER1 number, STRING1 Letter} ),
            letter, number);
```



```
ds3 := MERGE(ds1,ds2,SORTED(letter,number));  
SetDS := [ds1,ds2];  
ds4 := MERGE(SetDS,letter,number);
```

MERGEJOIN

MERGEJOIN(*setofdatabases*, *joincondition*, **SORTED**(*fields*) [*jointype*] [**DEDUP**] [**UNORDERED** | **ORDERED**(*bool*)] [**STABLE** | **UNSTABLE**] [**PARALLEL** [(*numthreads*)]] [**ALGORITHM**(*name*)])

<i>setofdatabases</i>	O SET de conjuntos de registro para processamento ([idx1,idx2,idx3]), normalmente INDEXes, em que todos precisam ter o mesmo formato.
<i>joincondition</i>	Uma expressão que especifica como corresponder registros no <i>setofdatabases</i> .
SORTED	Especifica a ordem de classificação de registros no <i>setofdatabases</i> de entrada, além da ordem de classificação de resultado no conjunto de resultados.
<i>fields</i>	Uma lista delimitada por vírgulas dos campos no <i>setofdatabases</i> , que precisa ser um subconjunto da ordem de classificação de entrada. Todos esses campos precisam ser usados na <i>joincondition</i> , uma vez que definem a ordem na qual os campos passam pela operação STEPPED.
<i>jointype</i>	Opcional. Se omitido, uma junção interna; caso contrário, um dos tipos listados abaixo.
DEDUP	Opcional. Especifica que o conjunto de resultados contém apenas registros únicos.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads.
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

A função **MERGEJOIN** é uma variação das formas do CONJUNTO DE DATASETS SET OF DATASETS das funções MERGE e JOIN. Assim como MERGE, ela funde os registros de *setofdatabases* em um único conjunto de resultados, porém como JOIN, ela usa a *joincondition* e *jointype* para determinar quais registros serão incluídos no conjunto de resultados. No entanto, ela não usa a função TRANSFORM para gerar o resultado; ela inclui todos os registros, inalterados, do *setofdatabases* correspondente a *joincondition*.

Lógica de Correspondência

Correspondência de registros *joincondition* pode conter duas partes: uma condição STEPPED que pode opcionalmente ser ANDed com condições sem STEPPED. A expressão STEPPED contém expressões de igualdade dos *campos* da opção SORTED, e ANDed, usando LEFT e RIGHT como qualificadores do dataset. Caso não esteja presente, a condição STEPPED é deduzida dos *campos* especificados pela opção SORTED.

A ordem dos datasets em *setofdatabases* pode ser importante para a maneira pela qual *joincondition* é avaliado. A *joincondition* é duplicada entre pares adjacentes de datasets, o que significa que essa *joincondition*:

LEFT.field = RIGHT.field

quando aplicada em um *setofdatabases* de três datasets, é logicamente equivalente a:

ds1.field = ds2.field AND ds2.field = ds3.field

Tipos de Join:

Usando os seguintes *jointypes* a seguir produzem os seguintes tipos de resultados com base na correspondência de registros produzida por *joincondition*:

INNER	Apenas os registros presentes em todos os datasets no <i>setofdatasets</i> .
LEFT OUTER	No mínimo um registro para cada registro no primeiro dataset no <i>setofdatasets</i> .
LEFT ONLY	Um registro para cada registro no primeiro dataset no <i>setofdatasets</i> para o qual não há correspondência em nenhum dos datasets subsequentes.
MOFN(min [,max])	Um registro para cada registro com registros correspondentes no número mínimo de datasets adjacentes no <i>setofdatasets</i> . Se o número máximo for especificado, o registro não será incluído se o número máximo de correspondências de datasets for excedido.

Exemplo:

```
Rec := RECORD,MAXLENGTH(4096)
  STRING1 Letter;
  UNSIGNED1 DS;
END;
ds1 := DATASET([{'A',1},{ 'B',1},{ 'C',1},{ 'D',1},{ 'E',1}],Rec);
ds2 := DATASET([{'A',2},{ 'B',2},{ 'H',2},{ 'I',2},{ 'J',2}],Rec);
ds3 := DATASET([{'B',3},{ 'C',3},{ 'M',3},{ 'N',3},{ 'O',3}],Rec);
ds4 := DATASET([{'A',4},{ 'B',4},{ 'R',4},{ 'S',4},{ 'T',4}],Rec);
ds5 := DATASET([{'B',5},{ 'V',5},{ 'W',5},{ 'X',5},{ 'Y',5}],Rec);
SetDS := [ds1,ds2,ds3,ds4,ds5];
j1 := MERGEJOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  SORTED(Letter));
j2 := MERGEJOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  SORTED(Letter),LEFT OUTER);
j3 := MERGEJOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  SORTED(Letter),LEFT ONLY);
j4 := MERGEJOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  SORTED(Letter),MOFN(3));
j5 := MERGEJOIN(SetDS,
  STEPPED(LEFT.Letter=RIGHT.Letter),
  SORTED(Letter),MOFN(3,4));
OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(j4);
OUTPUT(j5);
```

Ver também: MERGE, JOIN, STEPPED

MIN

MIN(*recordset*, *value* [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL**] [, (*numthreads*)] [, **ALGORITHM**(*name*)])

MIN(*valuelist*)

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Isto também pode ser a palavra-chave GROUP para indicar a localização do valor mínimo do campo em um grupo, quando usado com uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>value</i>	A expressão da qual o valor mínimo será encontrado.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
<i>valuelist</i>	Uma lista de expressões delimitada por vírgula da qual o valor mínimo será localizado. Também pode ser um SET de valores.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	MIN retorna um único valor.

A função **MIN** retorna o *valor* mínimo do *recordset* especificado ou o *valuelist*. Está configurada para retornar o valor zero caso o *recordset* esteja vazio.

Exemplo:

```
MinVal1 := MIN(Trades, Trades.trd_rate);
MinVal2 := MIN(4,8,16,2,1); //returns 1
SetVals := [4,8,16,2,1];
MinVal3 := MIN(SetVals); //returns 1
```

Ver também: **MAX**, **AVE**

NOLOCAL

NOLOCAL(*data*)

<i>data</i>	O nome de um atributo DATASET ou INDEX.
Return:	NOLOCAL retorna um conjunto de registros ou índice.

A função **NOLOCAL** especifica que todas as operações subsequentes nos *dados* são realizadas em todos os nós. Ela é geralmente usada em uma operação THISNODE . **Essa função está disponível para uso apenas no Roxie.**

Exemplo:

```
ds := JOIN(SomeData,NOLOCAL(SomeIndex), LEFT.ID = RIGHT.ID);
```

Ver também: ALLNODES, THISNODE, LOCAL

NONEMPTY

NONEMPTY(*recordsetlist*)

<i>recordsetlist</i>	Uma lista delimitada por vírgulas de conjuntos de registros.
Return:	NONEMPTY retorna um conjunto de registros.

A função **NONEMPTY** retorna o primeiro conjunto de registro do *recordsetlist* que contém quaisquer registros. Isso é semelhante ao uso da função EXISTS em uma expressão IF para retornar um de dois possíveis conjuntos de registros.

Exemplo:

```
ds := NONEMPTY(SomeData(SomeFilter),  
               SomeData(SomeOtherFilter),  
               SomeOtherData(YetAnotherFilter));
```

Ver também: EXISTS

NORMALIZE

NORMALIZE(*recordset*, *expression*, *transform* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

NORMALIZE(*recordset*, **LEFT**.*childdataset*, *transform* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento.
<i>expression</i>	Uma expressão numérica que especifica o número total de vezes para chamar a transformação para o registro.
<i>transform</i>	A função TRANSFORM para chamar para cada registro no recordset.
<i>childdataset</i>	O nome de campo de um DATASET secundário no recordset. É necessário usar a palavra-chave LEFT como qualificador.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	NORMALIZE retorna um conjunto de registros.

A função **NORMALIZE** normaliza os registros secundários de um *recordset* no qual os registros secundários estão anexados no fim dos registros de dados principais. O objetivo é pegar registros de arquivos simples e de comprimento variável, e dividir as informações secundárias. As informações primárias podem ser facilmente extraídas usando TABLE ou PROJECT.

Forma 1 do NORMALIZE

O form 1 é processada por todos os registros no *recordset*, desempenhando a função *transform* e a *expressão* em determinado número de vezes em cada registro por vez.

Requerimentos da Função TRANSFORM para a forma 1

A função *transform* precisa adotar no mínimo dois parâmetros: um registro LEFT do mesmo formato que o *recordset*, e um COUNTER de número inteiro especificando o número de vezes que a função *transform* foi acionado para esse registro. O formato de recordset resultante não precisa ser o mesmo da entrada.

Forma 2 do NORMALIZE

Form 2 processa todos os registros no *recordset* iterando a função *transform* em todos os registros *childdataset* em cada registro por vez.

Requerimentos da Função TRANSFORM para a forma 2

A função *TRANSFORM* deve adotar pelo menos um parâmetro: um registro *RIGHT* de mesmo formato que o *child-dataset*. O formato de recordset resultante não precisa ser o mesmo da entrada.

Exemplo:

```
//Form 1 example
NamesRec := RECORD

UNSIGNED1 numRows;
STRING20 thename;
STRING20 addr1 := '';
STRING20 addr2 := '';
STRING20 addr3 := '';
STRING20 addr4 := '';
END;

NamesTable := DATASET([ {1,'Kevin','10 Malt Lane'},
{2,'Liz','10 Malt Lane','3 The cottages'},
{0,'Mr Nobody'},
{4,'Anywhere','Here','There','Near','Far'}],
NamesRec);

OutRec := RECORD
UNSIGNED1 numRows;
STRING20 thename;
STRING20 addr;
END;

OutRec NormIt(NamesRec L, INTEGER C) := TRANSFORM
SELF := L;
SELF.addr := CHOOSE(C, L.addr1, L.addr2, L.addr3,
L.addr4);
END;

NormAddrs :=
    NORMALIZE(namesTable,LEFT.numRows,NormIt(LEFT,COUNTER));
/* the result is: numRows thename
addr
1 Kevin 10 Malt Lane
2 Liz 10 Malt Lane
2 Liz 3 The cottages
4 Anywhere Here
4 Anywhere There
4 Anywhere Near
4 Anywhere Far */
//*****
//Form 2 example
ChildRec := RECORD
INTEGER1 NameID;
STRING20 Addr;
END;

DenormedRec := RECORD
INTEGER1 NameID;
STRING20 Name;
DATASET(ChildRec) Children;
END;

ds := DATASET([ {1,'Kevin',[ {1,'10 Malt Lane'}]},
{2,'Liz', [ {2,'10 Malt Lane'},
{2,'3 The cottages'}]},
{3,'Mr Nobody', []},
{4,'Anywhere',[ {4,'Far'},
{4,'Here'}],
```



```
{4, 'There'},  
{4, 'Near'}}] } ],  
DenormedRec);  
ChildRec NewChildren(ChildRec R) := TRANSFORM  
SELF := R;  
END;  
NewChilds := NORMALIZE(ds, LEFT.Children, NewChildren(RIGHT));
```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, DENORMALIZE

NOFOLD

[*name* :=] **NOFOLD**(*expression*)

<i>name</i>	Opcional. O identificador desta função.
<i>expression</i>	A expressão a ser avaliada.

A função **NOFOLD** cria uma barreira que previne a ocorrência de otimizações entre a *expressão* e o contexto na qual ela é utilizada. Isso é usado para prevenir um dobramento constante no contexto para que ele possa ser avaliado no estado em que se encontra. Note que isso não previne o dobramento constante dentro da própria *expressão* . Normalmente, a função é usada apenas para evitar que os casos de teste sejam otimizados em algo completamente diferente ou para contornar temporariamente os erros no compilador.

Exemplo:

```
OUTPUT(2 * 2); // is normally constant folded to:
OUTPUT(4);     // at compile time.

//However adding NOFOLD() around one argument prevents that
OUTPUT(NOFOLD(2) * 2);

//Adding NOFOLD() around the entire expression does NOT
// prevent folding within the argument:
OUTPUT(NOFOLD(2 * 2));
//is the same as
OUTPUT(NOFOLD(4));
```

NOTHOR

[*name* :=] **NOTHOR**(*action*)

<i>name</i>	Opcional. O identificador desta ação.
<i>action</i>	A ação a ser executada.

A diretiva de compilador **NOTHOR** indica que a *ação* não deve ser executada no thor, mas em linha e em um contexto global. O NOTHOR permite realizar apenas operações de dataset bastante simples, como a filtragem de registros ou um simples PROJETO PROJECT.

O NOTHOR precisa ser usado em operações que usam as transações de superarquivo (como o exemplo abaixo), onde o compilador não reconhece o contexto apropriado.

Exemplo:

```
IMPORT STD;
rec := RECORD
  STRING10 S;
END;

srcnode := '10.239.219.2';
srcdir := '/var/lib/HPCCSystems/mydropzone/';

dir := STD.File.RemoteDirectory(srcnode,srcdir,'*.txt',TRUE);

//without NOTHOR this code gets this error:
// "Cannot call function AddSuperFile in a non-global context"
NOTHOR(SEQUENTIAL(
  STD.File.DeleteSuperFile('MultiSuper1'),
  STD.File.CreateSuperFile('MultiSuper1'),
  STD.File.StartSuperFileTransaction(),
  APPLY(dir,STD.File.AddSuperFile('MultiSuper1',
    STD.File.ExternalLogicalFileName(srcnode,srcdir+name))),
  STD.File.FinishSuperFileTransaction()));

F1 := DATASET('MultiSuper1', rec, THOR);
OUTPUT(F1,, 'testmultil',OVERWRITE);
```

Ver também: SEQUENTIAL

NOTIFY

[*attributename* :=] **NOTIFY**(*event* [, *parm*] [, *expression*])

<i>attributename</i>	Opcional. O identificador desta ação.
<i>event</i>	A função EVENT , ou uma constante de string sem distinção entre maiúsculas e minúsculas que nomeia o evento a ser gerado.
<i>parm</i>	Opcional. Uma constante de string sem distinção entre maiúsculas e minúsculas que contém o parâmetro do evento.
<i>expression</i>	Opcional. Uma constante de string sem distinção entre maiúsculas e minúsculas que permite a especificação simples da mensagem para restringir o evento a uma workunit específica.

A ação **NOTIFY** aciona o *evento* para que a função **WAIT** function ou o serviço de fluxo de trabalho **WHEN** possa proceder com as operações que estão encarregador de operar.

O parâmetro *expression* permite definir um serviço em ECL iniciado por um *evento*, respondendo apenas para a tarefa que o iniciou.

Exemplo:

```
NOTIFY('testevent', 'foobar');

receivedFileEvent(String name) := EVENT('ReceivedFile', name);
NOTIFY(receivedFileEvent('myfile'));

//as a service
doMyService := FUNCTION
OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
NOTIFY(EVENT('MyServiceComplete',
'<Event><returnTo>FRED</returnTo></Event>'),
EVENTEXTRA('returnTo'));
RETURN EVENTEXTRA('returnTo');
END;

doMyService : WHEN('MyService');
// and a call to the service
NOTIFY('MyService',
'<Event><returnTo>' + WORKUNIT + '</returnTo>...</Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

Ver também: **EVENT**, **EVENTNAME**, **EVENTEXTRA**, **CRON**, **WHEN**, **WAIT**

ORDERED

[*attributename* :=] **ORDERED**(*actionlist*)

<i>attributename</i>	Opcional. O nome da ação, que transforma a ação em definição de atributo, consequentemente não é executado até que <i>attributename</i> seja usado como uma ação.
<i>actionlist</i>	Uma lista delimitada por vírgula das ações a serem executadas na ordem. Podem ser ações ECL ou ações externas.

A ação **ORDERED** executa os itens da *actionlist* na ordem em que aparecem na *actionlist*. Isto é útil quando uma ação subsequente exige o resultado de uma ação precedente.

Possui os requisitos de ordenação de **SEQUENTIAL**. Esta não é a maneira mais útil de ordenar ações que não têm nada em comum, por exemplo, gerar arquivos e em seguida enviar e-mail. Se houver alguma chance de um valor compartilhado que possa mudar o significado, você deve usar **SEQUENTIAL**.

ORDERED não possui nenhum efeito sobre os atributos **PERSISTED**.

Exemplo:

```
Act1 :=  
    OUTPUT(A_People,OutputFormat1,'//hold01/fred.out');  
Act2 :=  
    OUTPUT(Person,{Person.per_first_name,Person.per_last_name})  
Act2 := OUTPUT(Person,{Person.per_last_name}));  
//by naming these actions, they become inactive  
    attributes  
//that only execute when the attribute names are called as  
    actions  
ORDERED(Act1,PARALLEL(Act2,Act3));  
//executes Act1 alone, and then executes Act2 and Act3 together
```

Ver também: **PARALLEL**, **PERSIST**, **SEQUENTIAL**

OUTPUT

[attr :=] **OUTPUT**(*recordset* [, *[format]* [, *[file [thorfileoptions]*]], **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*recordset*, *[format]*, *file*, **CSV** [(*csvoptions*)] [*csvfileoptions*] [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*recordset*, *[format]* , *file* , **XML** [(*xmloptions*)] [*xmlfileoptions*] [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*recordset*, *[format]* , *file* , **JSON** [(*jsonoptions*)] [*jsonfileoptions*] [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*recordset*, *[format]* , **PIPE**(*pipeoptions* [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)]] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*recordset* [, *format*] , **NAMED**(*name*) [, **EXTEND**] [, **ALL**] [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*expression* [, **NAMED**(*name*)] [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

[attr :=] **OUTPUT**(*recordset* , **THOR** [, **NOXPATH**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]);

<i>attr</i>	Opcional. O nome da ação transforma a ação em uma definição; consequentemente, não é executado até que <i>attr</i> seja usado como ação.
<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um record set derivado de algumas condições de filtro, ou qualquer expressão que resulte em um record set derivado.
<i>format</i>	Opcional. O formato dos registros de resultado. Se omitido, todos os campos em <i>recordset</i> serão resultados. Se não for omitido, deve ser o nome de uma definição de estrutura RECORD previamente definida ou um layout de registro "dinâmico" dentro de chaves ({}), e deve atender aos mesmos requisitos da estrutura RECORD para a função TABLE (a forma "corte vertical") definindo o tipo, nome e origem dos dados para cada campo.
<i>file</i>	Opcional. O nome lógico do arquivo no qual os registros serão gravados. Consulte a seção Escopo e Nomes de arquivos lógicos da Referência de Linguagem para obter mais detalhes sobre nomes de arquivos lógicos. Se omitido, o fluxo dos dados formatados retorna apenas para o emissor do comando (linha de comando ou IDE) e não é gravado em um arquivo de disco.
<i>thorfileoptions</i>	Opcional. Uma lista delimitada por vírgula das opções válidas para um arquivo THOR/FLAT (consulte a seção abaixo para obter mais detalhes).
NOXPATH	Especifica que qualquer XPATHs definido na estrutura <i>format</i> ou RECORD do <i>recordset</i> é ignorado, sendo substituídos pelos nomes do campo. Isso permite controlar se XPATHs estão sendo usados para obter resultados, assim os XPATHs específicos para entrada xml ou json podem ser ignorados para resultados.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for False, especifica que a ordem do registro de resultado não é importante. Quando for True, especifica a ordem padrão do registro de resultado.

Referência a Linguagem ECL
Ações e Funções Built-in

STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os threads <i>numthreads</i> .
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
CSV	Especifica que o arquivo é um arquivo ASCII delimitado por campo (geralmente com valores separados por vírgula).
<i>csvoptions</i>	Opcional. Uma lista delimitada por vírgula das opções que definem como o arquivo é delimitado.
<i>csvfileoptions</i>	Opcional. Uma lista delimitada por vírgula das opções válidas para um arquivo CSV (consulte a seção abaixo para obter mais detalhes).
XML	Especifica que o arquivo é produzido na forma de dados XML com o nome de cada campo no formato, tornando-se a tag XML para esses dados do campo.
<i>xmloptions</i>	Opcional. Uma lista separada por vírgula das opções que definem como o arquivo XML de resultado é delimitado.
<i>xmlfileoptions</i>	Opcional. Uma lista delimitada por vírgula das opções válidas para um arquivo XML (consulte a seção abaixo para obter mais detalhes).
JSON	Especifica que o arquivo é produzido na forma de dados JSON com o nome de cada campo no formato, tornando-se a tag JSON para esses dados do campo.
<i>jsonoptions</i>	Opcional. Uma lista separada por vírgula das opções que definem como o arquivo JSON de resultado é delimitado.
<i>jsonfileoptions</i>	Opcional. Uma lista delimitada por vírgula das opções válidas para um arquivo JSON (consulte a seção abaixo para obter mais detalhes).
PIPE:	Indica que o comando especificado é executado com o <i>recordset</i> fornecido como entrada padrão do comando. Este é um pipe de "gravação".
<i>pipeoptions</i>	O nome de um programa a ser executado, que toma o <i>arquivo</i> como seu fluxo de entrada, juntamente com as opções válidas para um PIPE de resultado.
NAMED	Especifica o nome do resultado que aparece na workunit. Não é válido se o parâmetro do arquivo estiver presente.
<i>name</i>	Uma constante de string que contém o rótulo do resultado. Deve ser uma constante de tempo de compilação e deve cumprir com os requisitos de nomeação de atributo.
EXTEND	Opcional. Especifica o acréscimo ao <i>nome</i> do resultado existente NAMED (NOMEADO) na workunit. O uso deste recurso exige que todos os OUTPUTs NAMED (RESULTADOS NOMEADOS) com o mesmo nome tenham a opção EXTEND disponível, incluindo a primeira instância.
ALL	Opcional. Especifica todos que todos os registros no <i>recordset</i> são enviados ao ECL IDE.
<i>expression</i>	Qualquer expressão ECL válida que resulta em um único valor escalar.
THOR	Especifica que o record set resultante é armazenado como um arquivo em disco “de propriedade” da workunit, em vez de ser armazenado diretamente dentro da workunit. O nome do arquivo no DFU é scope::RESULT::workunitid.

A ação **OUTPUT** produz como resultado um record set no supercomputador com base na forma e opções que você escolher. Se nenhum *arquivo* for especificado para gravação, o resultado é armazenado na workunit e retornado para o programa que o acionou na forma de fluxo de dados.

Nome dos Campos OUTPUT

Os nomes de campo em formato de registro "dinâmico" {...} devem ser exclusivos, caso contrário resultará em erro de sintaxe. Por exemplo:

```
OUTPUT(person(), {module1.attr1, module2.attr1});
```

resultará em um erro de sintaxe. Os nomes dos campos do resultado são adotados com base nos nomes de definição.

Para mudar isso, é possível especificar um nome exclusivo para o campo do resultado no formato de registro “dinâmico”, como este:

```
OUTPUT (person (), {module1.attr1, name: = module2.attr1});
```

OUTPUT Arquivos Thor/Flat

[*attr* :=] **OUTPUT**(*recordset* [, [*format*] [*file* [, **CLUSTER**(*target*)] [**ENCRYPT**(*key*)]
[**COMPRESSED**] [**OVERWRITE**] [, **UPDATE**] [**EXPIRE**([*days*])]]])

CLUSTER	Opcional. Especifica a gravação do arquivo em uma lista específica de clusters de destino. Se omitido, o arquivo é gravado no cluster pelo qual a workunit é executada. O número de partes do arquivo físico gravado em disco sempre é determinado pelo número de nós no cluster onde a workunit é executada, independentemente do número de nós nos clusters de destino.
<i>target</i>	Uma lista delimitada por vírgula das constantes de string que contêm os nomes dos clusters para os quais o arquivo será gravado. Os nomes devem estar listados como aparecem na página de Atividade do ECL Watch, ou como são retornados pela função Std.System.Thor-lib.Group(); opcionalmente, podem apresentar colchetes contendo uma lista delimitada por vírgula dos números dos nós (baseado em 1) e/ou dos intervalos (especificados com um traço, como p.ex., n-m) para indicar o conjunto específico de nós para gravar.
ENCRYPT	Opcional. Especifica a gravação do arquivo em disco usando a criptografia AES de 256 bits e a compactação LZW .
<i>key</i>	Uma constante de string que contém a chave de criptografia que será usada para criptografar os dados.
COMPRESSED	Opcional. Especifica a gravação do arquivo usando a compactação LZW.
OVERWRITE	Opcional. Especifica a substituição do arquivo caso ele exista.
UPDATE	Especifica que o arquivo deve ser regravado apenas se houver alteração nos dados de código ou de entrada.
EXPIRE	Opcional. Especifica que se trata de um arquivo temporário que pode ser removido automaticamente após um determinado número de dias, após a leitura ter sido feita.
<i>days</i>	Opcional. O número de dias contados a partir da última leitura do arquivo em que o arquivo será automaticamente removido. Se EXPIRE for especificado sem o número de dias, por padrão ele usará as configurações do ExpiryDefault no Sasha.

Esta forma grava o *record set* em um *arquivo* especificado no *formato* determinado. Se o *formato* for omitido, todos os campos do *recordset* serão considerados como resultados. Se *file* se o arquivo for omitido, o resultado é enviado de volta para o programa que o solicitou (geralmente o ECL IDE ou o programa que enviou a consulta SOAP para o Roxie).

Exemplo:

```
OutputFormat1 := RECORD
  People.firstname;
  People.lastname;
END;

A_People := People(lastname[1]='A');
Score1 := HASHCRC(People.firstname);
Attr1 := People.firstname[1] = 'A';

OUTPUT(SORT(A_People,Score1),OutputFormat1,'hold01:fred.out');
// writes the sorted A_People set to the fred.out file in
// the format declared in the OutputFormat1 definition

OUTPUT(People,{firstname,lastname});
// writes just First and Last Names to the command issuer
// full qualification of the fields is unnecessary, since
// the "on-the-fly" records structure is within the
// scope of the OUTPUT -- People is assumed

OUTPUT(People(Attr1=FALSE));
// writes all Peeople fields from records where Attr1 is
// false to the command issuer
```

Output de arquivos CSV

[*attr* :=] **OUTPUT**(*recordset*, [*format*] *file* , **CSV**[(*csvoptions*)] [, **CLUSTER**(*target*)] [**ENCRYPT**(*key*)] [**COMPRESSED**]

[**OVERWRITE**] [, **UPDATE**] [, **EXPIRE**([*days*])])

CLUSTER	Opcional. Especifica a gravação do arquivo em uma lista específica de clusters de destino. Se omitido, o arquivo é gravado no cluster pelo qual a workunit é executada. O número de partes do arquivo físico gravado em disco sempre é determinado pelo número de nós no cluster onde a workunit é executada, independentemente do número de nós nos clusters de destino.
<i>target</i>	Uma lista delimitada por vírgula das constantes de string que contêm os nomes dos clusters para os quais o arquivo será gravado. Os nomes devem estar listados como aparecem na página de Atividade do ECL Watch, ou como são retornados pela função Std.System.Thorlib.Group(); opcionalmente, podem apresentar colchetes contendo uma lista delimitada por vírgula dos números dos nós (baseado em 1) e/ou dos intervalos (especificados com um traço, como p.ex., n-m) para indicar o conjunto específico de nós para gravar.
ENCRYPT	Opcional. Especifica a gravação do arquivo em disco usando a criptografia AES de 256 bits e a compactação LZW .
<i>key</i>	Uma constante de string que contém a chave de criptografia que será usada para criptografar os dados.
COMPRESSED	Opcional. Especifica a gravação do arquivo usando a compactação LZW.
OVERWRITE	Opcional. Especifica a substituição do arquivo caso ele exista.
UPDATE	Especifica que o arquivo deve ser regravado apenas se houver alteração nos dados de código ou de entrada.
EXPIRE	Opcional. Especifica que se trata de um arquivo temporário que pode ser removido automaticamente após um determinado número de dias.
<i>days</i>	Opcional. O número de dias em que o arquivo será automaticamente removido. Se omitido, o padrão é sete (7).

Esta forma grava o *recordset* no *arquivo* especificado e no *formato* determinado na forma de arquivo ASCII com valores separados por vírgula. Um conjunto válido de *csvoptions* é:

HEADING([*headertext* [, *footertext*]] [, **SINGLE**] [, **FORMAT**(*stringfunction*)])

SEPARATOR(*delimiters*)

TERMINATOR(*delimiters*)

QUOTE([*delimiters*])

ASCII | **EBCDIC** | **UNICODE**

HEADING	Especifica os cabeçalhos e os rodapés do arquivo.
<i>headertext</i>	Opcional. O texto do registro de cabeçalho a ser colocado no arquivo. Se omitido, os nomes do campo serão utilizados.
<i>footertext</i>	Opcional. O texto do registro de rodapé a ser colocado no arquivo. Se omitido, nenhum <i>footertext</i> será gerado.
SINGLE	Opcional. Especifica o uso da <i>headertext</i> especifica que o texto do cabeçalho é gravado apenas no início da parte 1 e o <i>footertext</i> é gravado apenas no final da parte n (gerando um arquivo CSV “padrão”). Se omitido, o padrão é: <i>headertext</i> e <i>footertext</i> se omitido, o texto de cabeçalho e o texto de rodapé são colocados no início e no fim de cada parte do arquivo (útil para gerar resultados XML complexos).
FORMAT	Opcional. Especifica que o texto de cabeçalho deve ser formatado usando <i>stringfunction</i> .
<i>stringfunction</i>	Opcional. A função usada para formatar os títulos (cabeçalhos) da coluna. Pode ser qualquer função que toma um único parâmetro de string e retorna um resultado da string.
SEPARATOR	Especifica os delimitadores do campo.
<i>delimiters</i>	Uma constante de string única (ou uma lista de constantes de string delimitada por vírgula) que define os caracteres usados para delimitar os dados no arquivo CSV.
TERMINATOR	Especifica os delimitadores do registro.
QUOTE	Especifica os <i>delimitadores</i> do texto dos valores da string que podem conter <i>delimitadores</i> SEPARATOR ou TERMINATOR como parte dos seus dados.
ASCII	Especifica que todos os resultados estão em formato ASCII, incluindo quaisquer campos EBCDIC ou UNICODE.
EBCDIC	Especifica que todos os resultados estão em formato EBCDIC, exceto SEPARATOR e TERMINATOR (que são definidos como valores ASCII).
UNICODE	Especifica que todos os resultados estão em formato Unicode UTF8

Se nenhuma das opções ASCII, EBCDIC, ou UNICODE forem especificadas, o resultado padrão será no formato ASCII com quaisquer campos UNICODE no formato UTF8. As outras *csvoptions* padrão são:

```
CSV(HEADING(' ',' '), SEPARATOR(','), TERMINATOR('\n'), QUOTE())
```

Exemplo:

```
//SINGLE option writes the header only to the first file part:
OUTPUT(ds, '~thor::outdata.csv', CSV(HEADING(SINGLE)));

//This example writes the header and footer to every file part:
OUTPUT(XMLds, '~thor::outdata.xml', CSV(HEADING('<XML>', '</XML>')));

//FORMAT option writes the header using the specified formatting function:
```

```
IMPORT STD;  
OUTPUT(ds, '~thor::outdata.csv', CSV(HEADING(FORMAT(STD.Str.ToUpperCase))));
```

Output de Arquivos XML

[*attr* :=] **OUTPUT**(*recordset*, [*format*], *file* , **XML** [(*xmloptions*)] [, **ENCRYPT**(*key*)] [, **CLUSTER**(*target*)] [, **OVERWRITE**] [, **UPDATE**] [, **EXPIRE**([*days*])])

CLUSTER	Opcional. Especifica a gravação do arquivo em uma lista específica de clusters de destino. Se omitido, o arquivo é gravado no cluster pelo qual a workunit é executada. O número de partes do arquivo físico gravado em disco sempre é determinado pelo número de nós no cluster onde a workunit é executada, independentemente do número de nós nos clusters de destino.
<i>target</i>	Uma lista delimitada por vírgula das constantes de string que contêm os nomes dos clusters para os quais o arquivo será gravado. Os nomes devem estar listados como aparecem na página de Atividade do ECL Watch, ou como são retornados pela função Std.System.Thorlib.Group(); opcionalmente, podem apresentar colchetes contendo uma lista delimitada por vírgula dos números dos nós (baseado em 1) e/ou dos intervalos (especificados com um traço, como p.ex., n-m) para indicar o conjunto específico de nós para gravar.
ENCRYPT	Opcional. Especifica a gravação do arquivo em disco usando a criptografia AES de 256 bits e a compactação LZW .
<i>key</i>	Uma constante de string que contém a chave de criptografia que será usada para criptografar os dados.
OVERWRITE	Opcional. Especifica a substituição do arquivo caso ele exista.
UPDATE	Especifica que o arquivo deve ser regravado apenas se houver alteração nos dados de código ou de entrada.
EXPIRE	Opcional. Especifica que se trata de um arquivo temporário que pode ser removido automaticamente após um determinado número de dias.
<i>days</i>	Opcional. O número de dias em que o arquivo será automaticamente removido. Se omitido, o padrão é sete (7).

Esta forma grava o *recordset* em um *arquivo* específico na forma de dados XML com o nome de cada campo no *formato* especificado, tornando-se a tag XML para esses dados do arquivo. O conjunto válido de *xmloptions* é:

'rowtag'

HEADING(*headertext* [, *footertext*])

TRIM

OPT

<i>rowtag</i>	O texto a ser incluído na tag de delimitação do registro.
HEADING	Especifica a inclusão dos registros de cabeçalho e rodapé no arquivo.
<i>headertext</i>	O texto do registro de cabeçalho a ser colocado no arquivo.
<i>footertext</i>	O texto do registro de rodapé a ser colocado no arquivo.
TRIM	Especifica a remoção dos espaços em branco dos campos da string antes da produção do resultado
OPT	Especifica as tags omitidas em qualquer campo de string vazio do resultado.

Se nenhuma *xmloptions* for especificada, o padrão é:

```
XML('Row',HEADING('<Dataset>\n','</Dataset>\n'))
```

Exemplo:

```
R := {STRING10 fname,STRING12 lname};
B := DATASET([{'Fred','Bell'},{'George','Blanda'},{'Sam',''}],R);

OUTPUT(B,,'fred1.xml',XML); // writes B to the fred1.xml file
/* the Fred1.XML file looks like this:
<Dataset>
  <Row><fname>Fred </fname><lname>Bell</lname></Row>
  <Row><fname>George</fname><lname>Blanda </lname></Row>
  <Row><fname>Sam </fname><lname></lname></Row>
</Dataset> */

OUTPUT(B,,'fred2.xml',XML('MyRow', HEADING('<?xml version=1.0 ...?>
  \n<filetag>\n','</filetag>\n')));
/* the Fred2.XML file looks like this:
<?xml version=1.0 ...?>
<filetag>
  <MyRow><fname>Fred </fname><lname>Bell</lname></MyRow>
  <MyRow><fname>George</fname><lname>Blanda</lname></MyRow>
  <MyRow><fname>Sam </fname><lname></lname></MyRow>
</filetag> */

OUTPUT(B,,'fred3.xml',XML('MyRow',TRIM,OPT));
/* the Fred3.XML file looks like this:
<Dataset>
  <MyRow><fname>Fred</fname><lname>Bell</lname></MyRow>
  <MyRow><fname>George</fname><lname>Blanda</lname></MyRow>
  <MyRow><fname>Sam</fname></MyRow>
</Dataset> */
```

Output de Arquivos Json

[*attr* :=] **OUTPUT**(*recordset*, [*format*], *file* ,**JSON** [(*jsonoptions*)] [**ENCRYPT**(*key*)] [**CLUSTER**(*target*)] [**OVERWRITE**] [**UPDATE**] [**EXPIRE**([*days*])])

CLUSTER	Opcional. Especifica a gravação do arquivo em uma lista específica de clusters de destino. Se omitido, o arquivo é gravado no cluster pelo qual a workunit é executada. O número de partes do arquivo físico gravado em disco sempre é determinado pelo número de nós no cluster onde a workunit é executada, independentemente do número de nós nos clusters de destino.
<i>target</i>	Uma lista delimitada por vírgula das constantes de string que contêm os nomes dos clusters para os quais o arquivo será gravado. Os nomes devem estar listados como aparecem na página de Atividade do ECL Watch, ou como são retornados pela função Std.System.Thorlib.Group(); opcionalmente, podem apresentar colchetes contendo uma lista delimitada por vírgula dos números dos nós (baseado em 1) e/ou dos intervalos (especificados com um traço, como p.ex., n-m) para indicar o conjunto específico de nós para gravar.
ENCRYPT	Opcional. Especifica a gravação do arquivo em disco usando a criptografia AES de 256 bits e a compactação LZW .
<i>key</i>	Uma constante de string que contém a chave de criptografia que será usada para criptografar os dados.
OVERWRITE	Opcional. Especifica a substituição do arquivo caso ele exista.
UPDATE	Especifica que o arquivo deve ser regravado apenas se houver alteração nos dados de código ou de entrada.
EXPIRE	Opcional. Especifica que se trata de um arquivo temporário que pode ser removido automaticamente após um determinado número de dias.

<i>days</i>	Opcional. O número de dias em que o arquivo será automaticamente removido. Se omitido, o padrão é sete (7).
-------------	---

Esta forma grava o *recordset* em um *arquivo* específico na forma de dados JSON com o nome de cada campo no *formato* especificado, tornando-se a tag JSON para esses dados do arquivo. O conjunto válido de *jsonoptions* é:

'rowtag'

HEADING(*headertext* [, *footertext*])

TRIM

OPT

<i>rowtag</i>	O texto a ser incluído na tag de delimitação do registro.
HEADING	Especifica a inclusão dos registros de cabeçalho e rodapé no arquivo.
<i>headertext</i>	O texto do registro de cabeçalho a ser colocado no arquivo.
<i>footertext</i>	O texto do registro de rodapé a ser colocado no arquivo.
TRIM	Especifica a remoção dos espaços em branco dos campos da string antes da produção do resultado
OPT	Especifica as tags omitidas em qualquer campo de string vazio do resultado.

Se nenhum *jsonoptions* for especificado, o padrão é:

```
JSON('Row',HEADING('[' , ' ']))
```

Exemplo:

```
R := {STRING10 fname,STRING12 lname};
B := DATASET([{'Fred','Bell'},{'George','Blanda'},{'Sam',''}],R);

OUTPUT(B,,'fred1.json',JSON); // writes B to the fred1.json file
/* the Fred1.json file looks like this:
{"Row": [
{"fname": "Fred", "lname": "Bell"},
{"fname": "George", "lname": "Blanda"},
{"fname": "Sam", "lname": ""}
]}
*/
OUTPUT(B,,'fred2.json',JSON('MyResult',HEADING('[' , ' ')));
/* the Fred2.json file looks like this:
["MyResult": [
{"fname": "Fred", "lname": "Bell"},
{"fname": "George", "lname": "Blanda"},
{"fname": "Sam", "lname": ""}
]]
```

Output de Arquivos PIPE

[*attr* :=] **OUTPUT**(*recordset*, [*format*] ,**PIPE**(*command* [, **CSV** | **XML**]) [, **REPEAT**])

PIPE	Indica que o comando especificado é executado com o recordset fornecido como entrada padrão do comando. Este é um pipe de "gravação".
<i>command</i>	O nome de um programa a ser executado que usa o arquivo como seu fluxo de entrada.
CSV	Opcional. Especifica que o formato dos dados de resultado é CSV. Se omitido, o formato será raw.

XML	Opcional. Especifica que o formato dos dados de resultado é em XML. Se omitido, o formato será raw.
REPEAT	Opcional. Indica uma nova instância que o comando especificado executa para cada linha no conjunto de registros.

Esta forma envia o *recordset* no *formato* especificado na forma de entrada padrão para o *comando*. Isso é comumente conhecido como "pipe de resultado."

Exemplo:

```
OUTPUT(A_People,,PIPE('MyCommandLineProgram'),OVERWRITE);  
    // sends the A_People to MyCommandLineProgram as  
    // standard in
```

Output Named

[*attr* :=] **OUTPUT**(*recordset* [,*format*],**NAMED**(*name*) [,**EXTEND**] [,**ALL**])

Esta forma grava o *recordset* na workunit com o *nome* especificado. A opção **EXTEND** permite várias ações **OUTPUT** em um mesmo resultado *nomeado* . A opção **ALL** é usada para substituir **CHOOSSEN** implícita aplicada às consultas interativas no Programa do Compilador de Consultas ECL. Isso especifica o retorno de todos os registros.

Exemplo:

```
OUTPUT(CHOOSSEN(people(firstname[1]='A'),10));  
    // writes the A People to the query builder  
OUTPUT(CHOOSSEN(people(firstname[1]='A'),10),ALL);  
    // writes all the A People to the query builder  
OUTPUT(CHOOSSEN(people(firstname[1]='A'),10),NAMED('fred'));  
    // writes the A People to the fred named output  
  
//a NAMED, EXTEND example:  
errMsgRec := RECORD  
    UNSIGNED4 code;  
    STRING text;  
END;  
makeErrMsg(UNSIGNED4 _code,STRING _text) := DATASET([{_code, _text}], errMsgRec);  
rptErrMsg(UNSIGNED4 _code,STRING _text) := OUTPUT(makeErrMsg(_code,_text),  
    NAMED('ErrorResult'),EXTEND);  
  
OUTPUT(DATASET([{100, 'Failed'}],errMsgRec),NAMED('ErrorResult'),EXTEND);  
    //Explicit syntax.  
  
//Something else creates the dataset  
OUTPUT(makeErrMsg(101, 'Failed again'),NAMED('ErrorResult'),EXTEND);  
  
//output and dataset handled elsewhere.  
rptErrMsg(102, 'And again');
```

Output Valores Scalar

[*attr* :=] **OUTPUT**(*expression* [, **NAMED**(*name*)])

Esta forma é usada para permitir o envio da *expressão* escalar, especialmente dentro das ações **PARALLEL**.

Exemplo:

```
OUTPUT(10) // scalar value output  
OUTPUT('Fred') // scalar value output
```

Output Arquivos de Workunit

`[attr :=] OUTPUT(recordset , THOR)`

Esta forma é usada para armazenar o *recordset* resultante na forma de arquivo em disco "de propriedade" da workunit. O nome do arquivo na DFU é *scope::RESULT::workunitid*. Isso é útil quando é preciso exibir um *recordset* de resultado grande no programa Compilador de Consultas ECL, porém sem que os dados ocupem memória no armazenamento de dados do sistema.

Exemplo:

```
OUTPUT(Person(per_st='FL'), THOR)
// output records to screen, but store the
// result on disk instead of in the workunit
```

Ver também: TABLE, DATASET, PIPE, CHOSEN

PARALLEL

[*definitionname* :=] **PARALLEL**(*actionlist*)

<i>definitionname</i>	Opcional. O nome da ação, que transforma a ação em uma definição, consequentemente não é executado até que <i>definitionname</i> seja usado como uma ação.
<i>actionlist</i>	Uma lista delimitada por vírgula das ações a serem executadas simultaneamente. Podem ser ações ECL ou ações externas.

A ação **PARALLEL** permite que os itens na *actionlist* sejam executados simultaneamente. Ela não força uma execução paralela, apenas a permite – o compilador determina a ordem real de execução. Este já é um modo de operação padrão, portanto **PARALLEL** só é útil dentro da lista de ação de um conjunto de ações **SEQUENTIAL**.

Exemplo:

```
Act1 :=  
OUTPUT(A_People,OutputFormat1,'//hold01/fred.out');  
Act2 :=  
OUTPUT(Person,{Person.per_first_name,Person.per_last_name});  
  
Act2 := OUTPUT(Person,{Person.per_last_name});  
  
//by naming these actions, they become inactive definitions  
//that only execute when the definition names are called as actions  
  
SEQUENTIAL(Act1,PARALLEL(Act2,Act3));  
  
//executes Act1 alone, and only when it's finished,  
// executes Act2 and Act3 together
```

Ver também: **ORDERED**, **SEQUENTIAL**

PARSE

PARSE(*dataset*, *data*, *pattern*, *result* , *flags* [, **MAXLENGTH**(*length*)])

PARSE(*dataset*, *data*, *result* , **XML**(*path*) [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>dataset</i>	O conjunto de registros para processamento.
<i>data</i>	Uma expressão que especifica o texto a ser interpretado (analisado), tipicamente o nome de um campo em um dataset.
<i>pattern</i>	O padrão de análise a ser correspondido.
<i>result</i>	O nome do atributo da estrutura RECORD que especifica o formato do conjunto de registro de resultado (como a função TABLE), ou a função TRANSFORM que gera o conjunto do registro de resultado (como PROJECT).
<i>flags</i>	Uma ou mais das opções de análise listadas abaixo.
MAXLENGTH	Especifica o comprimento máximo que o padrão pode corresponder. Se omitido, o comprimento padrão é 4096.
<i>length</i>	Uma constante inteira que especifica o número máximo de caracteres correspondentes.
XML	Especifica que o dataset contém dados XML.
<i>path</i>	Uma constante da string que contém XPATH na tag que delimita os dados XML no dataset.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads.
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	PARSE retorna um conjunto de registros.

A função **PARSE** desempenha um texto ou uma operação de análise XML .

PARSE de Dados Textuais

A primeira forma opera sobre o *dataset*, localizando registros cujos *dados* possuem um *padrão correspondente* e gerando um conjunto de resultados dessas correspondências no formato de *resultado* . Se o *padrão* localizar múltiplas correspondências nos *dados*, um registro de resultado será gerado para cada uma delas. Cada correspondência para PARSE é efetivamente um único caminho através do *padrão*. Se houver mais de um caminho correspondente, quando o *resultado* transform é acionado uma vez para cada caminho ou quando a opção BEST é usada, o caminho com a penalidade mais baixa será selecionado.

Se o *resultado* nomear uma estrutura RECORD , essa forma de PARSE funcionará como a função TABLE para gerar o conjunto de resultado, porém também funcionará em texto de comprimento variável. Se o *resultado* nomear uma

Referência a Linguagem ECL

Ações e Funções Built-in

função TRANSFORM , a função “transform” irá gerar o conjunto de resultado. A função TRANSFORM deve adotar pelo menos um parâmetro: um registro LEFT de mesmo formato que o *dataset*. O formato do conjunto de registro resultante não precisa ser o mesmo do da entrada.

Indicadores indicadores podem ter os seguintes valores:

FIRST	Retorna uma linha apenas para a primeira correspondência que inicia em uma posição específica.
ALL	Retorna uma linha para cada possível correspondência da string em uma posição específica.
WHOLE	Corresponde apenas à string completa.
NOSCAN	Se houver uma posição correspondente, não continua a busca por outras correspondências.
SCAN	Se houver uma posição correspondente, continua a busca a partir do final da correspondência ou a partir da próxima posição.
SCAN ALL	Retorna correspondências para cada posição inicial possível. Use a função TRIM para eliminar a análise de espaços em branco estranhos.
NOCASE	Realiza uma comparação sem distinção de maiúsculas e minúsculas.
CASE	Realiza uma comparação com distinção de maiúsculas e minúsculas (esse é o padrão).
SKIP (separator-pattern)	Especifica um padrão que pode ser inserido após cada token em um padrão de busca. Por exemplo, SKIP ([' '\t']*) pula espaços e guias entre os tokens.
KEEP (max)	Mantém apenas a primeira correspondência <i>max</i> .
ATMOST (max)	Não gera nenhuma correspondência se houver mais de uma <i>correspondências max</i> .
MAX	Retorna uma linha para o resultado que corresponde à sequência mais longa da entrada. Apenas uma correspondência é retornada, a menos que a opção MANY tenha sido especificada.
MIN	Retorna uma linha para o resultado que corresponde à sequência mais curta da entrada. Apenas uma correspondência é retornada, a menos que a opção MANY tenha sido especificada.
MATCHED ([rule-reference])	Usada quando <i>referência de regra</i> é usada em uma função de correspondência de usuário. Se a referência de regra não for especificada, as informações correspondentes podem não ser preservadas.
MATCHED (ALL)	Mantém todos os nomes de regra – se forem usados pelas funções de correspondência de usuário.
NOT MATCHED	Gera uma linha se não houver correspondências na linha de entrada. Todos os acionamentos da função MATCHED() retornarão como falsos dentro da <i>resultstructure</i> .
NOT MATCHED ONLY	Gera uma linha apenas se nenhuma correspondência for encontrada.
BEST	Seleciona a correspondência com maior pontuação (penalidade mais baixa). Se os indicadores MAX ou MIN também estiverem presentes, eles serão ativados primeiro. Apenas uma correspondência é retornada, a menos que a opção MANY tenha sido especificada.
MANY	Retornar várias correspondências para as opções MELHOR, MÁX ou MÍN.
PARSE	Implementa a análise Tomita Implementa análise em vez da tecnologia de análise de expressão regular.
USE ([struct,] x)	Especifica usando um atributo de padrão RULE definido mais adiante no código com a função DEFINE(x), introduzindo uma gramática recursiva (a única recursão permitida no ECL) Se a <i>estrutura</i> RECORD opcional for especificada, USE especifica utilizando o atributo de padrão RULE definido mais adiante no código com a função DEFINE(x) que

gera um resultado de linha no formato de *estrutura* RECORD (válido apenas se a opção PARSE também estiver presente). USE é exigido em PARSE quando qualquer padrão não puder ser localizado ao conduzir as regras a partir da raiz sem seguir nenhum USEs.

Exemplo:

```
rec := {STRING10000 line};
datafile := DATASET([
  {'Ge 34:2 And when Shechem the son of Hamor the Hivite, prince of the country, saw her,'+
    ' he took her, and lay with her, and defiled her.'},
  {'Ge 36:10 These are the names of Esaus sons; Eliphaz the son of Adah the wife of Esau,'+
    ' Reuel the son of Bashemath the wife of Esau.'}],rec);
PATTERN ws1 := [' ','\t',' ',''];
PATTERN ws := ws1 ws1?;
PATTERN patStart := FIRST | ws;
PATTERN patEnd := LAST | ws;
PATTERN article := ['A','The','Thou','a','the','thou'];

TOKEN patWord := PATTERN('[a-zA-Z]+');
TOKEN Name := PATTERN('[A-Z][a-zA-Z]+');

RULE Namet := name OPT(ws ['the','king of','prince of'] ws name);
PATTERN produced := OPT(article ws) ['begat','father of','mother of'];
PATTERN produced_by := OPT(article ws) ['son of','daughter of'];
PATTERN produces_with := OPT(article ws) ['wife of'];

RULE relationtype := ( produced | produced_by | produces_with);
RULE progeny := namet ws relationtype ws namet;

results := RECORD
  STRING60 Le := MATCHTEXT(Namet[1]);
  STRING60 Ri := MATCHTEXT(Namet[2]);
  STRING30 RelationPhrase := MatchText(relationtype);
END;
outfile1 := PARSE(datafile,line,progeny,results,SCAN ALL);
```

PARSE de dados XML

A segunda forma funciona a partir de um *dataset* XML, analisando os *dados* XML e criando um recordset com o uso do parâmetro *resultado*, um registro de resultado por entrada. A expectativa é que cada linha de *dados* contenha um bloco completo de XML. Se o *resultado* nomear uma estrutura RECORD, essa forma de PARSE funcionará como a função TABLE para gerar o recordset.

Se o *resultado* nomear uma função TRANSFORM, a função “transform” irá gerar o conjunto de resultado. A função TRANSFORM deve adotar pelo menos um parâmetro: um registro LEFT de mesmo formato que o *dataset*. O formato do conjunto de registro resultante não precisa ser o mesmo do da entrada.

OBSERVAÇÃO: A leitura e a análise do XML podem consumir muita memória dependendo do uso. Em particular, se o xpath especificado combinar com uma quantidade muito grande de dados, uma grande estrutura de dados será então fornecida para transformação. Dessa forma, quanto maior for a combinação, mais recursos serão consumidos por combinação. Por exemplo, se você possui um documento muito grande e combinar um elemento próximo da raiz que, virtualmente, engloba o documento todo, o documento inteiro será interpretado como uma estrutura referenciável que o ECL terá acesso.

Exemplo:

```
linerec := { STRING line };
in1 := DATASET([{
  '<ENTITY eid="P101" type="PERSON" subtype="MILITARY">' +
  ' <ATTRIBUTE name="fullname">JOHN SMITH</ATTRIBUTE>' +
```

```
' <ATTRIBUTE name="honorific">Mr.</ATTRIBUTE>' +
' <ATTRIBUTEGRP descriptor="passport">' +
'   <ATTRIBUTE name="idNumber">W12468</ATTRIBUTE>' +
'   <ATTRIBUTE name="idType">pp</ATTRIBUTE>' +
'   <ATTRIBUTE name="issuingAuthority">JAPAN PASSPORT AUTHORITY</ATTRIBUTE>' +
'   <ATTRIBUTE name="country" value="L202"/>' +
'   <ATTRIBUTE name="age" value="19"/>' +
' </ATTRIBUTEGRP>' +
'</ENTITY>' }],
  linerec);
passportRec := RECORD
  STRING id;
  STRING idType;
  STRING issuer;
  STRING country;
  INTEGER age;
END;
outrec := RECORD
  STRING id;
  UNICODE fullname;
  UNICODE title;
  passportRec passport;
  STRING line;
END;
outrec t(lineRec L) := TRANSFORM
  SELF.id := XMLTEXT('@eid');
  SELF.fullname := XMLUNICODE('ATTRIBUTE[@name="fullname"]');
  SELF.title := XMLUNICODE('ATTRIBUTE[@name="honorific"]');
  SELF.passport.id := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
    + '/ATTRIBUTE[@name="idNumber"]');
  SELF.passport.idType := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
    + '/ATTRIBUTE[@name="idType"]');
  SELF.passport.issuer := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
    + '/ATTRIBUTE[@name="issuingAuthority"]');
  SELF.passport.country := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
    + '/ATTRIBUTE[@name="country"]/@value');
  SELF.passport.age := (INTEGER)XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
    + '/ATTRIBUTE[@name="age"]/@value');
  SELF := L;
END;

textout := PARSE(in1, line, t(LEFT), XML('/ENTITY[@type="PERSON"]'));
```

Ver também: DATASET, OUTPUT, XMLENCODE, XMLDECODE, REGEXFIND, REGEXREPLACE, DEFINE

Exemplos de PARSE Estendido

Este exemplo analisa os números de telefone não processados de um campo específico em um dataset de entrada em um único resultado padrão contendo apenas os números. O código de área ausente nos dados de entrada não processados resulta em três zeros iniciais no resultado.

```
infile := DATASET(['5619994581'], {'15619994581'},
  {'(561) 999-4581'}, {'(561)999-4581'},
  {'561-999-4581'}, {'561 999 4581'},
  {'561.999.4581'}, {'561/999/4581'},
  {'561 999-4581'}, {'9994581'},
  {'999-4581'}], {STRING20 rawnumber});

PATTERN numbers := PATTERN('[0-9]')+;
PATTERN alpha := PATTERN('[A-Za-z]')+;
PATTERN ws := [' ', '\t']*;
PATTERN sepchar := PATTERN('[-./ ]');
```

```
PATTERN Seperator := ws sepchar ws;

// Area Code
PATTERN OpenParen := ['(','{','<'];
PATTERN CloseParen := [']','}','>'];
PATTERN FrontDigit := ['1', '0'] OPT(Seperator);
PATTERN areacode := OPT(FrontDigit) OPT(OpenParen) numbers length(3) OPT(CloseParen);

// Last Seven digits
PATTERN exchange := numbers length(3);
PATTERN lastfour := numbers length(4);
PATTERN seven := exchange OPT(Seperator) lastfour;

// Extension
PATTERN extension := ws alpha ws numbers;

// Phone Number
PATTERN phonenumber := OPT(areacode) OPT(Seperator) seven
    opt(extension) ws;

layout_phone_append := RECORD
    infile;
    STRING10 clean_phone := MAP(NOT MATCHED(phonenumber) => '',
        NOT MATCHED(areacode) => '000' + MATCHTEXT(exchange) + MATCHTEXT(lastfour),
        MATCHTEXT(areacode/numbers) + MATCHTEXT(exchange) + MATCHTEXT(lastfour));
END;

outfile :=
    PARSE(infile, rawnumber, phonenumber, layout_phone_append, FIRST, NOT MATCHED, WHOLE);

OUTPUT(outfile);
```

Este exemplo analisa um subconjunto pequeno de dados não processados de filmes (disponíveis gratuitamente em IMDB.com) em campos de base de dados padrão:

```
Layout_Actors_Raw := RECORD
    STRING120 IMDB_Actor_Desc;
END;

File_Actors := DATASET([
    {'A.V., Subba Rao Chenchu Lakshmi (1958/I) <10>'},
    {' Jayabheri (1959) <17>'},
    {' Madalasa (1948) <3>'},
    {' Mangalya Balam (1958) <12>'},
    {' Mohini Bhasmasura (1938) <3>'},
    {' Palletoori Pilla (1950) [Kampanna Dora] <4>'},
    {' Peddamanushulu (1954) <6>'},
    {' Sarangadhara (1957) <12>'},
    {' Sri Seetha Rama Kalyanam (1961) <12>'},
    {' Sri Venkateswara Mahatmyam (1960) [Akasa Raju] <5>'},
    {' Vara Vikrayam (1939) [Judge] <12>'},
    {' Vindhyanani (1948) <7>'},
    {' '},
    {'Aa, Brynjar Adjo solidaritet (1985) [Ponker] <40>'},
    {' '},
    {'Aabel, Andreas Bor Borson Jr. (1938) [O.G. Hansen] <9>'},
    {' Jeppe pa bjerget (1933) [En skomakerlaerling]'},
    {' Kampen om tungtvannet (1948) <8>'},
    {' Prinsessen som ingen kunne maqlbinde (1932) [Eспен
        Askeladd] <3>'},
    {' Spokelse forelsker seg, Et (1946) [Et spokelse] <6>'},
    {' '},
    {'Aabel, Hauk (I) Alexander den store (1917) [Alexander Nyberg]'},
    {' Du har lovet mig en kone! (1935) [Professoren] <6>'},
    {' Glad gutt, En (1932) [Ola Nordistua] <1>'}
```

```
{ ' Jeppe pa bjerget (1933) [Jeppe] <1>',
  ' Morderen uten ansikt (1936)' },
{ ' Store barnedapen, Den (1931) [Evensen, kirketjener] <5>',
  ' Troll-Elgen (1927) [Piper, direktor] <9>',
  ' Ungen (1938) [Krestoffer] <8>',
  ' Valfangare (1939) [Jensen Sr.] <4>',
  '' },
{ ' Aabel, Per (I) Brudebuketten (1953) [Hoyland jr.] <3>',
  ' Cafajestes, Os (1962)' },
{ ' Farlige leken, Den (1942) [Fredrik Holm, doktor]' },
{ ' Herre med bart, En (1942) [Ole Grong, advokat] <1>',
  ' Kjaere Maren (1976) [Doktor]' },
{ ' Kjaerlighet og vennskap (1941) [Anton Schack] <3>',
  ' Ombyte fornojer (1939) [Gregor Ivanow] <2>',
  ' Portrettet (1954) [Per Haug, provisor] <1>' }],
Layout_Actors_Raw);

//Basic patterns:
PATTERN arb := PATTERN('[-!.,\t a-zA-Z0-9]')+;

//all alphanumeric & certain special characters
PATTERN ws := [' ', '\t']+; //word separators (space & tab)
PATTERN number := PATTERN('[0-9]')+; //numbers

//extended patterns:
PATTERN age := '(' number OPT('/I') ')';

//movie year -- OPT('/I') required for first rec
PATTERN role := '[' arb ']'; //character played
PATTERN m_rank := '<' number '>'; //credit appearance number
PATTERN actor := arb OPT(ws '(I)' ws);
//actor's name -- OPT(ws '(I)' ws)
// required for last two actors

//extended pattern to parse the actual text:
PATTERN line := actor '\t' arb ws OPT(age) ws OPT(role) ws OPT(m_rank) ws;

//output record structure:
NLP_layout_actor_movie := RECORD
  STRING30 actor_name := Std.Str.filterout(MATCHTEXT(actor), '\t');
  STRING50 movie_name := MATCHTEXT(arb[2]);
  UNSIGNED2 movie_year := (UNSIGNED)MATCHTEXT(age/number);
  STRING20 movie_role := MATCHTEXT(role/arb);
  UNSIGNED1 cast_rank := (UNSIGNED)MATCHTEXT(m_rank/number);
END;

//and the actual parsing operation
Actor_Movie_Init := PARSE(File_Actors,
                          IMDB_Actor_Desc,
                          line,
                          NLP_layout_actor_movie, WHOLE, FIRST);

// then iterate to propagate actor name in each record
NLP_layout_actor_movie IterNames(NLP_layout_actor_movie L,
                                NLP_layout_actor_movie R) := TRANSFORM
  SELF.actor_name := IF(R.actor_Name='', L.actor_Name, R.actor_name);
  SELF:= R;
END;

NLP_Actor_Movie := ITERATE(Actor_Movie_Init, IterNames(LEFT, RIGHT));

// and output the result set
OUTPUT(NLP_Actor_Movie);
```

PIPE

PIPE(*command*, *recorddef* [, **CSV** | **XML**])

PIPE(*recordset*, *command* [, *recorddef*] [, **REPEAT**] [, **CSV** | **XML**] [, **OUTPUT**(**CSV** | **XML**)] [, **GROUP**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>command</i>	O nome do programa a ser executado, o qual precisa usar quaisquer dados de entrada pelo stdin e produzir seu resultado pelo stdout. Esse programa já precisa ter sido implementado no cluster do HPCC no diretório de instância do Thor (como: /var/lib/HPCCSystems/mythor/), mas pode ser substituído pela configuração de ambiente externalProgDir para o cluster Thor).
<i>recorddef</i>	O formato da estrutura RECORD do resultado. Se omitido, o resultado será o mesmo que o formato de entrada.
CSV	Opcional. No form 1 (e como o parâmetro para a opção OUTPUT), especifica que o formato de dados de resultado é CSV. No form 2, especifica que o formato de dados de entrada é CSV. Se omitido, o formato será raw.
XML	Opcional. No form 1 (e como o parâmetro para a opção OUTPUT), especifica que o formato de dados de resultado é XML. No form 2, especifica que o formato de dados de entrada é XML. Se omitido, o formato será raw.
<i>recordset</i>	O dataset de entrada.
REPEAT	Opcional. Especifica que uma nova instância do programa de comando é criada para cada linha no recordset.
OUTPUT	Opcional. Especifica o formato de dados do resultado CSV ou XML.
GROUP	Opcional. Especifica que cada registro de resultado é gerado em um GROUP separado (apenas se REPEAT for especificado).
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for "False" (Falso), especifica que a ordem do registro de resultado não é importante. Quando for "True" (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	PIPE retorna um conjunto de registros.

A função **PIPE** permite que o código ECL inicie um programa de *comando* externo em cada nó, paralelizando efetivamente um programa de processamento não paralelo. O PIPE tem duas formas:

A forma 1 não recebe entradas, executa o *comando* e gera o resultado no formato *recorddef*. Esse é um pipe de "entrada" (como a opção PIPE em uma definição DATASET).

A forma 2 recebe a entrada *recordset*, executa o *comando* e gera o resultado no formato *recorddef*. Esse é um pipe de "passagem".

Exemplo:

```
namesRecord := RECORD
  STRING10 forename;
  STRING10 surname;
  STRING2 nl := '\r\n';
END;

d := PIPE('pipeRead 200', namesRecord); //form 1 - input pipe

t := PIPE(d, 'pipeThrough'); //form 2 - through pipe

OUTPUT(t,,PIPE('pipeWrite \\thordata\\names.all')); //output pipe

//Form 2 with XML input:
namesRecord := RECORD
  STRING10 Firstname{xpath('/Name/FName')};
  STRING10 Lastname{xpath('/Name/LName')};
END;

p := PIPE('echo <Name><FName>George</FName><LName>Jetson</LName></Name>', namesRecord, XML);
OUTPUT(p);
```

Ver também: OUTPUT, DATASET

POWER

POWER(*base*,*exponent*)

<i>base</i>	O número real a ser elevado.
<i>exponent</i>	A potência real na qual x será elevado.
Return:	POWER retorna um valor real único.

A função **POWER** retorna o resultado da *base* elevada à potência do *exponent* .

Exemplo:

```
MyCube := POWER(2.0,3.0); // = 8  
MySquare := POWER(3.0,2.0); // = 9
```

Ver também: SQRT, EXP, LN

PRELOAD

PRELOAD(*file* [, *nbr*])

<i>file</i>	O nome de uma definição DATASET
<i>nbr</i>	Opcional. Uma constante inteira especificando quantos índices devem ser criados “instantaneamente” para acelerar o acesso ao arquivo do DATASET especificado (apenas). Se for > 1.000, especifica a quantidade de memória reservada para esses índices.
Retorno:	PRELOAD retorna um conjunto de registros.

A função **PRELOAD** deixa o *arquivo* na memória após ser carregado (válido apenas para uso do Motor de entrega rápida de dados). Isso é equivalente ao uso da opção PRELOAD na definição DATASET.

Exemplo:

```
MyFile := DATASET('MyFile',{STRING20 F1, STRING20 F2},THOR);  
COUNT(PRELOAD(MyFile))
```

Ver também: DATASET

PROCESS

PROCESS(*recordset*, *datarow*, *datasettransform*, *rowtransform* [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** ((*numthreads*))] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento.
<i>datarow</i>	O registro RIGHT inicial para processamento, normalmente expresso pela função ROW.
<i>datasettransform</i>	A função TRANSFORM a ser acionada para cada registro no recordset.
<i>rowtransform</i>	A função TRANSFORM a ser acionada para gerar o próximo registro RIGHT do <i>datasettransform</i> .
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os threads <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	PROCESS retorna um conjunto de registros.

A função **PROCESS** opera de maneira similar à **ITERATE** no sentido de processar todos os registros no *recordset* (um par de registros por vez), realizando a função *datasettransform* em cada par de registros por vez. O primeiro registro no recordset é especificado para o *datasettransform* como o primeiro registro esquerdo, emparelhado com o *datarow* como o registro direito. O *rowtransform* é usado para gerar o registro direito para o próximo par. Se o *datasettransform* ou o *rowtransform* contiver **SKIP**, então nenhum registro será gerado pelo *datasettransform* para o registro ignorado.

Requerimentos da Função TRANSFORM - PROCESS

Ambas as funções *datasettransform* e *rowtransform* precisam de no mínimo dois parâmetros: um registro LEFT no mesmo formato que o *recordset* e um registro RIGHT no mesmo formato que o *datarow*. O formato do recordset resultante para o *datasettransform* precisa ser idêntico àquele do *recordset* de entrada. O formato do recordset resultante para o *rowtransform* precisa ser idêntico àquele do *datarow* inicial. Opcionalmente, o *datasettransform* pode usar um terceiro parâmetro: um COUNTER de número inteiro para especificar o número de vezes que a função transform foi acionada para o *recordset* ou para o grupo atual no *recordset* (consulte a função **GROUP**).

Exemplo:

```
DSrec := RECORD
  STRING4 Letter;
  STRING4 LeftRecIn := '';
  STRING4 RightRecIn := '';
END;
```

```
StateRec := RECORD
  STRING2 Letter;
END;
ds := DATASET([{'AA'},{'BB'},{'CC'},{'DD'},{'EE'}],DSrec);

DSrec DSxform(DSrec L,StateRec R) := TRANSFORM
  SELF.Letter := L.Letter[1..2] + R.Letter;
  SELF.LeftRecIn := L.Letter;
  SELF.RightRecIn := R.Letter;
END;
StateRec ROWxform(DSrec L,StateRec R) := TRANSFORM
  SELF.Letter := L.Letter[1] + R.Letter[1];
END;

p := PROCESS(ds,
              ROW({'ZZ'},StateRec),
              DSxform(LEFT,RIGHT),
              ROWxform(LEFT,RIGHT));

OUTPUT(p);
/* Result:
AAZZ AA ZZ
BBAZ BB AZ
CCBA CC BA
DDCB DD CB
EEDC EE DC */

//*****
// This examples uses different information for state tracking
// (the point of the PROCESS function) through the input record set.

w1 := RECORD
  STRING v{MAXLENGTH(100)};
END;

s1 := RECORD
  BOOLEAN priorA;
END;

ds := DATASET([{'B'},{'A'}, {'C'}, {'D'}], w1);

s1 doState(w1 l, s1 r) := TRANSFORM
  SELF.priorA := l.v = 'A';
END;

w1 doRecords(w1 l, s1 r) := TRANSFORM
  SELF.v := l.v + IF(r.priorA, '****', '');
END;

initState := ROW({TRUE}, s1);

rs := PROCESS(ds,
              initState,
              doRecords(LEFT,RIGHT),
              doState(LEFT,RIGHT));

OUTPUT(rs);
/* Result:
B***
A
C***
D
*/
```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, ROW, ITERATE

PROJECT

PROJECT(*recordset*, *transform* [, **PREFETCH** [(*lookahead* [, **PARALLEL**)]] [, **KEYED**] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

PROJECT(*recordset*, *record* [, **PREFETCH** [(*lookahead* [, **PARALLEL**)]] [, **KEYED**] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Esse pode ser um DATASET em linha de registro único.
<i>transform</i>	A função TRANSFORM a ser acionada para cada registro no recordset.
PREFETCH	Opcional. Permite que leituras de índice dentro do transform sejam tão eficientes quanto JOINS com chave. Válido apenas para consultas ECL no Roxie.
<i>lookahead</i>	Opcional. Especifica o número de leituras antecipadas. Se omitido, o padrão é o valor da tag <code>_PrefetchProjectPreload</code> na consulta enviada. Se for omitido, o valor de <code>defaultPrefetchProjectPreload</code> especificado no arquivo <code>RoxieTopology</code> será usado quando o Roxie foi implantado. Se for omitido, 10 será usado por padrão.
PARALLEL	Opcional. Especifica que a consulta avançada é feita em um thread separado, em paralelo com a execução da consulta.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
<i>record</i>	A estrutura RECORD do resultado estrutura para uso em cada registro no recordset.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	PROJECT retorna um conjunto de registros.

A função **PROJECT** processa todos os registros no *recordset*, realizam a função *transform* em cada registro por vez.

A forma 1 **PROJECT**(*recordset*,*record*) é basicamente um sinônimo abreviado para:

PROJECT(*recordset*,**TRANSFORM**(*record*,**SELF** := **LEFT**)).

simplicando a transferência de dados de uma estrutura para outra sem um **TRANSFORM**, contanto que todos os campos na estrutura de *registro* de resultado estejam presentes no *recordset* de entrada.

Requisitos da Função TRANSFORM - PROJECT

A função *transform* precisa usar no mínimo um parâmetro: um registro LEFT de mesmo formato que o *recordset*. Opcionalmente, ele pode usar um segundo parâmetro: um COUNTER inteiro que especifique o número de vezes que a função *transform* foi acionada para o *recordset*, ou outro grupo atual no *recordset* (consulte a função GROUP). O segundo form de parâmetro é útil para adicionar números de sequência. O formato do conjunto de registro resultante não precisa ser o mesmo do da entrada.

Exemplo:

```
//form one example *****
Ages := RECORD
  STRING15 per_first_name;
  STRING25 per_last_name;
  INTEGER8 Age;
END;
TodaysYear := 2001;

Ages CalcAges(person l) := TRANSFORM
  SELF.Age := TodaysYear - l.birthdate[1..4];
  SELF := l;
END;
AgedRecs := PROJECT(person, CalcAges(LEFT));

//COUNTER example *****
SequencedAges := RECORD
  Ages;
  INTEGER8 Sequence := 0;
END;

SequencedAges AddSequence(Ages l, INTEGER c) :=
  TRANSFORM
    SELF.Sequence := c;
    SELF := l;
END;
SequencedAgedRecs := PROJECT(AgedRecs,
  AddSequence(LEFT,COUNTER));

//form two example *****
NewRec := RECORD
  STRING15 firstname;
  STRING25 lastname;
  STRING15 middlename;
END;
NewRecs := PROJECT(People,NewRec);
//equivalent to:
//NewRecs := PROJECT(People,TRANSFORM(NewRec,SELF :=
  LEFT));

//LOCAL example *****
MyRec := RECORD
  STRING1 Value1;
  STRING1 Value2;
END;

SomeFile := DATASET([{'C','G'},{'C','C'},{'A','X'},
  {'B','G'},{'A','B'}],MyRec);

MyOutRec := RECORD
  SomeFile.Value1;
  SomeFile.Value2;
```

```

    STRING6 CatValues;
END;

DistFile := DISTRIBUTE(SomeFile,HASH32(Value1,Value2));

MyOutRec CatThem(SomeFile L, INTEGER C) := TRANSFORM
    SELF.CatValues := L.Value1 + L.Value2 + '-' +
                      (Std.System.Thorlib.Node()+1) + '-' + (STRING)C;
    SELF := L;
END;

CatRecs := PROJECT(DistFile,CatThem(LEFT,COUNTER),LOCAL);

OUTPUT(CatRecs);

/* CatRecs result set is:
Rec# Value1 Value2 CatValues
1      C      C      CC-1-1
2      B      G      BG-2-1
3      A      X      AX-2-2
4      A      B      AB-3-1
5      C      G      CG-3-2
*/

```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, ROW, DATASET

PROJECT - Módulo

PROJECT(*module*, *interface* [, **OPT** | *attributelist*])

<i>module</i>	A estrutura MODULE que contém definições de atributo cujos valores passam como a interface.
<i>interface</i>	A estrutura INTERFACE para passar.
OPT	Opcional. Suprime a mensagem de erro que é gerada quando um atributo definido na interface não é definido também no módulo.
<i>attributelist</i>	Opcional. Uma lista de atributos específicos delimitada por vírgula no módulo para fornecer à interface. Isso permite a implementação de uma lista especificada de atributos, algo que é útil se você quiser um controle mais rígido ou se os tipos de parâmetros não corresponderem.
Return:	PROJECT retorna um MODULE compatível com a interface.

A função **PROJECT** passa os atributos de um *módulo* na forma de *interface* para uma função definida para aceitar parâmetros estruturados como a *interface* especificada. Isso permite que você crie um módulo para uma *interface* com os valores fornecidos por outra interface. Os atributos no *módulo* precisam ser compatíveis com os atributos na *interface* (do mesmo tipo e mesmos parâmetros, se usar algum parâmetro).

Exemplo:

```

PROJECT(x,y)
/*is broadly equivalent to
MODULE(y)
    SomeAttributeInY := x.someAttributeInY
    //... repeated for all attributes in Y ...
END;
*/

myService(myInterface myArgs) := FUNCTION
    childArgs := MODULE(PROJECT(myArgs,Iface,isDead,did,ssn,address))
    BOOLEAN isFCRA := myArgs.isFCRA OR myArgs.fakeFCRA
END;
RETURN childService(childArgs);

```

```
END;  
  
// you could directly pass PROJECT as a module parameter  
// to an attribute:  
myService(myInterface myArgs) := childService(PROJECT(myArgs, childInterface));
```

Ver também: Estrutura MODULE, Estrutura INTERFACE, Estrutura FUNCTION, STORED

PULL

PULL(*dataset*)

<i>dataset</i>	O conjunto de registros que será completamente carregado na Refinaria de Dados.
Return:	PULL retorna um recordset.

A função **PULL** é uma meta-operação projetada apenas para sugerir que o *dataset* deve ser totalmente carregado na Refinaria de Dados antes de continuar a operação na Refinaria de Dados.

Exemplo:

```
MySet := PULL(Person);  
//load Person into Data Refinery before continuing
```

Ver também:

RANDOM

RANDOM()

Return:	RANDOM retorna um único valor.
---------	--------------------------------

A função **RANDOM** retorna um valor inteiro pseudo-aleatório e não negativo entre 0 e 4.294.967.295.

Exemplo:

```
MySet := DISTRIBUTE(Person,RANDOM()); //random distribution
```

Ver também: DISTRIBUTE

RANGE

RANGE(*setofddatasets*, *setofintegers* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>setofddatasets</i>	Um conjunto de datasets.
<i>setofintegers</i>	Um conjunto de valores inteiros.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	RANGE retorna um conjunto de datasets.

A função **RANGE** extrai um subconjunto do *setofddatasets* na forma de um SET. O *setofintegers* especifica quais elementos do *setofddatasets* compreendem o SET resultante de datasets. Isto é normalmente usado na função **GRAPH**.

Exemplo:

```
r := {STRING1 Letter};
ds1 := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],r);
ds2 := DATASET([{'F'},{'G'},{'H'},{'I'},{'J'}],r);
ds3 := DATASET([{'K'},{'L'},{'M'},{'N'},{'O'}],r);
ds4 := DATASET([{'P'},{'Q'},{'R'},{'S'},{'T'}],r);
ds5 := DATASET([{'U'},{'V'},{'W'},{'X'},{'Y'}],r);

SetDS := [ds1,ds2,ds3,ds4,ds5];
outDS := RANGE(setDS,[1,3]);
//use only 1st and 3rd elements

OUTPUT(outDS[1]); //results in A,B,C,D,E
OUTPUT(outDS[2]); //results in K,L,M,N,O
```

Ver também: **GRAPH**

RANK

RANK(*position*, *set* [, **DESCEND**])

<i>position</i>	Um valor inteiro que indica o elemento a ser retornado do conjunto classificado.
<i>set</i>	O conjunto de valores.
DESCEND	Opcional. Indica a classificação de ordem decrescente.
Return:	RANK retorna um único valor.

A função **RANK** classifica o *conjunto* em ordem ascendente (ou decrescente se **DESCEND** estiver presente), e retorna a posição ordinal (o valor do índice) do elemento de *posição* do *conjunto* classificado no conjunto não classificado. Trata-se do oposto de **RANKED**.

Exemplo:

```
Ranking := RANK(1,[20,30,10,40]);  
// returns 2 - 1st element (20) in unsorted set is  
// 2nd element after sorting to [10,20,30,40]  
Ranking := RANK(1,[20,30,10,40],DESCEND);  
// returns 3 - 1st element (20) in unsorted set is  
// 3rd element after sorting to [40,30,20,10]
```

Ver também: **RANKED**, **SORT**, **SORTED**, Conjuntos e Filtros

RANKED

RANKED(*position*, *set* [, **DESCEND**])

<i>position</i>	Um valor inteiro que indica o elemento a ser retornado do conjunto não classificado.
<i>set</i>	O conjunto de valores.
DESCEND	Opcional. Indica a classificação de ordem decrescente.
Return:	RANKED retorna um único valor.

A função **RANKED** classifica o *conjunto* em ordem ascendente (ou decrescente se **DESCEND** estiver presente), e retorna a posição ordinal (o valor do índice) do elemento de *posição* do conjunto classificado no *conjunto* não classificado. Trata-se do oposto de **RANK**.

Exemplo:

```
Ranking := RANKED(1,[20,30,10,40]);  
// returns 3 - 1st element (10) in sorted set [10,20,30,40]  
// was 3rd element in unsorted set  
  
Ranking := RANKED(1,[20,30,10,40],DESCEND);  
// returns 4 - 1st element (40) in sorted set [40,30,20,10]  
// was 4th element in unsorted set
```

Ver também: **RANK**, **SORT**, **SORTED**, Conjuntos e Filtros

REALFORMAT

REALFORMAT(*expression*, *width*, *decimals*)

<i>expression</i>	A expressão que especifica o valor REAL a ser formatado.
<i>width</i>	O tamanho da string na qual o valor será alinhado à direita.
<i>Decimais</i>	Um valor inteiro que especifica o número de casas decimais.
Return:	REALFORMAT retorna um único valor.

A função **REALFORMAT** retorna o valor da *expressão* formatada como uma string justificada à direita de caracteres de *largura* com o número de *decimais* especificados.

Exemplo:

```
REAL8 Float := 1000.0063;  
STRING12 FloatStr12 := REALFORMAT(float,12,6);  
OUTPUT(FloatStr12); //results in ' 1000.006300'
```

Ver também: INTFORMAT

REGEXFIND

REGEXFIND(*regex*, *text* [, *flag*] [, **NOCASE**])

<i>regex</i>	Uma expressão Perl regular padrão.
<i>text</i>	O texto a ser analisado.
<i>flag</i>	Opcional. Especifica o texto a ser retornado. Se omitido, REGEXFIND retorna TRUE ou FALSE caso <i>regex</i> tenha ou não sido localizado no texto. Se for 0, a parte do texto em que <i>regex</i> foi correspondido será retornada. Se for >= 1, o texto correspondido pelo grupo nth em <i>regex</i> será retornado.
NOCASE	Opcional. Especifica uma busca sem distinção entre maiúsculas e minúsculas.
Return:	REGEXFIND retorna um único valor.

A função **REGEXFIND** usa *regex* para analisar o *texto* e localizar correspondências. A *regex* deve ser uma expressão regular Perl padrão. Usamos bibliotecas de terceiros para suportar isso. Portanto, para texto de código único consulte os documentos Boost em *text* http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html. Observe que a versão da biblioteca Boost pode variar de acordo com sua distribuição. Para *texto* unicode, consulte os documentos ICU nas seções “Regular Expression Metacharacters” (Metacaracteres de expressão regular) e “Regular Expression Operators” (Operadores de expressão regular) em <http://userguide.icu-project.org/strings/regexp> e os links indicados lá, especificamente a seção “UnicodeSet patterns” (Padrões UnicodeSet) em <http://userguide.icu-project.org/strings/unicodeset>. Nós usamos a versão 2.6, que deve suportar todos os recursos listados.

Exemplo:

```
namesRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  STRING10 userdate;
END;
namesTbl := DATASET([ {'Halligan','Kevin','10/14/1998'},
  {'Halligan','Liz','12/01/1998'},
  {'Halligan','Jason','01/01/2000'},
  {'MacPherson','Jimmy','03/14/2003'} ],
  namesRecord);
searchpattern := '^(.*)/(.*)/(.*)$';
search := '10/14/1998';

filtered := namesTbl(REGEXFIND('^(Mc|Mac)', surname));

OUTPUT(filtered); //1 record -- MacPherson
OUTPUT(namesTbl,{(string30)REGEXFIND(searchpattern,userdate,0),
  (string30)REGEXFIND(searchpattern,userdate,1),
  (string30)REGEXFIND(searchpattern,userdate,2),
  (string30)REGEXFIND(searchpattern,userdate,3)});

REGEXFIND(searchpattern, search, 0); //returns
  '10/14/1998'
REGEXFIND(searchpattern, search, 1); //returns '10'
REGEXFIND(searchpattern, search, 2); //returns '14'
REGEXFIND(searchpattern, search, 3); //returns '1998'
```

Ver também: PARSE, REGEXFINDSET, REGEXREPLACE

REGEXFINDSET

REGEXFINDSET(*regex*, *text* [, **NOCASE**])

<i>regex</i>	Uma expressão Perl regular padrão.
<i>text</i>	O texto a ser analisado.
NOCASE	Opcional. Especifica uma busca sem distinção entre maiúsculas e minúsculas.
Return:	REGEXFINDSET retorna um conjunto de strings.

A função **REGEXFIND** usa *regex* para analisar o *texto* e localizar correspondências. A regex deve ser uma expressão regular Perl padrão.. Usamos bibliotecas de terceiros para suportar isso. Portanto, para *texto* de código único consulte os documentos Boost em http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html Observe que a versão da biblioteca Boost pode variar de acordo com sua distribuição. Para *texto* unicode, consulte os documentos ICU nas seções “Regular Expression Metacharacters” (Metacaracteres de expressão regular) e “Regular Expression Operators” (Operadores de expressão regular) em <http://userguide.icu-project.org/strings/regex> e os links indicados lá, especificamente a seção “UnicodeSet patterns” (Padrões UnicodeSet) em <http://userguide.icu-project.org/strings/unicodeset>. Nós usamos a versão 2.6, que deve suportar todos os recursos listados.

Exemplo:

```
sampleStr :=
  'To: jane@example.com From: john@example.com This is the winter of our discontent.';
eMails:=REGEXFINDSET('\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}', sampleStr);
OUTPUT(eMails);

UNICODE sampleStr2:=
  U'To: janě@example.com From john@example.com This is the winter of our discontent.';
eMails2:= REGEXFINDSET(U'\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}', sampleStr2);
OUTPUT(eMails2);
```

Ver também: PARSE, REGEXFIND, REGEXREPLACE

REGEXREPLACE

REGEXREPLACE(*regex*, *text*, *replacement* [, **NOCASE**])

<i>regex</i>	Uma expressão Perl regular padrão.
<i>text</i>	O texto a ser analisado.
<i>replacement</i>	O texto de substituição. Nesta string, \$0 refere-se à substring que correspondeu com o padrão <i>regex</i> , e \$1, \$2, \$3... corresponde com o primeiro, segundo, terceiro... grupos no padrão.
NOCASE	Opcional. Especifica uma busca sem distinção entre maiúsculas e minúsculas.
Return:	REGEXREPLACE retorna um único valor.

A função **REGEXREPLACE** usa *regex* para analisar o *texto* e localizar correspondências, a fim de em seguida substituí-las com a string de *substituição*. A regex deve ser uma expressão regular Perl padrão. Usamos bibliotecas de terceiros para suportar isso. Portanto, para texto de código único consulte os documentos Boost em http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html. Observe que a versão da biblioteca Boost pode variar de acordo com sua distribuição. Para *texto* unicode, consulte os documentos ICU nas seções “Regular Expression Metacharacters” (Metacaracteres de expressão regular) e “Regular Expression Operators” (Operadores de expressão regular) em <http://userguide.icu-project.org/strings/regexp> e os links indicados lá, especificamente a seção “UnicodeSet patterns” (Padrões UnicodeSet) em <http://userguide.icu-project.org/strings/unicodeset>. Nós usamos a versão 2.6, que deve suportar todos os recursos listados.

Exemplo:

```
REGEXREPLACE('(.)t', 'the cat sat on the mat', '$1p');
//ASCII
REGEXREPLACE(u'(.a)t', u'the cat sat on the mat', u'$1p');
//UNICODE
// both of these examples return 'the cap sap on the map'

inrec := {STRING10 str, UNICODE10 ustr};
inset := DATASET([{'She', u'Eins'}, {'Sells', u'Zwei'},
{'Sea', u'Drei'}, {'Shells', u'Vier'}], inrec);
outrec := {STRING10 orig, STRING10 withcase, STRING10
wocase,
UNICODE10 uorig, UNICODE10 uwithcase, UNICODE10 uwocase};

outrec trans(inrec l) := TRANSFORM
SELF.orig := l.str;
SELF.withcase := REGEXREPLACE('s', l.str, 'f');
SELF.wocase := REGEXREPLACE('s', l.str, 'f', NOCASE);
SELF.uorig := l.ustr;
SELF.uwithcase := REGEXREPLACE(u'e', l.ustr, u'\u00EB');
SELF.uwocase := REGEXREPLACE(u'e', l.ustr, u'\u00EB',
NOCASE);
END;
OUTPUT(PROJECT(inset, trans(LEFT)));

/* the result set is:
orig withcase wocase uorig uwithcase uwocase
She She fhe Eins Eins \xc3\xabins
Sells Sellf fellf Zwei Zw\xc3\xabi Zw\xc3\xabi
Sea Sea fea Drei Dr\xc3\xabi Dr\xc3\xabi
Shells Shellf fhellf Vier Vi\xc3\xabr Vi\xc3\xabr */
```

Ver também: PARSE, REGEXFIND

REGROUP

REGROUP(*recset*,...,*recset* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL**] (*numthreads*)] [, **ALGORITHM**(*name*)])

<i>recset</i>	Um conjunto agrupado de registros. Cada <i>recset</i> deve ser exatamente do mesmo tipo e deve conter o mesmo número de grupos.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	REGROUP retorna um conjunto de registros.

A função **REGROUP** combina os *recsets* agrupados em um único conjunto de registros agrupado. Isto é obtido combinando cada grupo no primeiro *recset* com os grupos na mesma posição ordinal dentro de cada *recset* subsequente.

Exemplo:

```
inrec := {UNSIGNED6 did};

outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;

ds := DATASET([1,2,3,4,5,6], inrec);
dsg := GROUP(ds, ROW);

i1 := DATASET([
  {1, 'Kevin', 10},
  {2, 'Richard', 5},
  {5, 'Nigel', 2},
  {0, '', 0}], outrecrec);
i2 := DATASET([
  {1, 'Kevin Halligan', 12},
  {2, 'Ricardo Chapman', 15},
  {3, 'Jake Smith', 20},
  {5, 'David Hicks', 100},
  {0, '', 0}], outrecrec);
i3 := DATASET([
  {1, 'Halligan', 8},
  {2, 'Ricardo', 8},
  {6, 'Pete', 4},
  {6, 'Peter', 8},
  {6, 'Petie', 1},
  {0, '', 0}], outrecrec);

j1 := JOIN(dsg, i1, LEFT.did = RIGHT.did, LEFT OUTER, MANY LOOKUP);
j2 := JOIN(dsg, i2, LEFT.did = RIGHT.did, LEFT OUTER, MANY LOOKUP);
j3 := JOIN(dsg, i3, LEFT.did = RIGHT.did, LEFT OUTER, MANY LOOKUP);
```

```
combined := REGROUP(j1, j2, j3);  
OUTPUT(j1);  
OUTPUT(j2);  
OUTPUT(j3);  
OUTPUT(combined);
```

Ver também: GROUP, COMBINE

REJECTED

REJECTED(*condition*,...,*condition*)

<i>condition</i>	Uma expressão condicional para avaliar.
Return:	REJECTED retorna um único valor.

A função **REJECTED** avalia quais das listas de *condições* retornaram como “false” (falsas) e retorna sua posição ordinal na lista de *condições*. Se nenhuma retornar como falsa, é indicado zero (0). Trata-se do oposto da função **WHICH**.

Exemplo:

```
Rejects := REJECTED(Person.first_name <> 'Fred',  
Person.first_name <> 'Sue');  
// Rejects receives 0 for everyone except those named Fred or Sue
```

Ver também: **WHICH**, **MAP**, **CHOOSE**, **IF**, **CASE**

ROLLUP

ROLLUP(*recordset*, *condition*, *transform* [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

ROLLUP(*recordset*, *transform*, *fieldlist* [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

ROLLUP(*recordset*, **GROUP**, *transform* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O recordset a ser processado, normalmente classificado na mesma ordem em que a condição ou <i>fieldlist</i> será testada.
<i>condition</i>	Uma expressão que define registros “duplicados”. As palavras-chave LEFT e RIGHT podem ser usadas como qualificadores de dataset nos campos do <i>recordset</i> .
<i>transform</i>	A função TRANSFORM é usada para acionar cada par de registros duplicados encontrado.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
<i>fieldlist</i>	Uma lista delimitada por vírgula de expressões ou campos no conjunto de registros que define os registros “duplicados”. As palavras-chave WHOLE RECORD (ou apenas RECORD) devem ser usadas para indicar todos os campos nessa estrutura, e/ou você pode usar a palavra-chave EXCEPT para listar os campos a serem excluídos.
GROUP	Especifica que o <i>recordset</i> é GROUPed e a operação ROLLUP irá gerar um único registro de resultado para cada grupo. Se não for este o caso, ocorrerá um erro.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	ROLLUP retorna um conjunto de registros.

A função **ROLLUP** é semelhante à função DEDUP , porém possui o acionamento da função *transform* para processar cada par de registro duplicado. Isso permite recuperar informações valiosas do registro “duplicado” antes que ele seja descartado. Dependendo de como você codifica o *transform* , ROLLUP pode manter o registro LEFT ou RIGHT , ou qualquer mistura de dados de ambos.

A primeira forma de ROLLUP testa uma condição usando valores dos registros que seriam especificados como LEFT e RIGHT para *transform*. Os registros são combinados se a condição for true (verdadeira). A segunda forma de ROLLUP compara valores de registros adjacentes no *recordset* de entrada, combinando-os caso sejam iguais. Estas duas formas adotarão um comportamento diferente se *transform* modificar alguns dos campos usados na condição de correspondência (veja os exemplos abaixo).

Para o primeiro par de registros candidatos, o registro LEFT especificado para “transform” corresponde ao primeiro registro do par, e o registro RIGHT o segundo. Para as correspondências subsequentes de mesmo valor, o registro LEFT especificado corresponde ao registro de resultado do acionamento anterior do *transform*, e o registro RIGHT o próximo registro no *recordset*, como descrito neste exemplo:

```
ds := DATASET([ {1,10}, {1,20}, {1,30}, {3,40}, {4,50} ],
              {UNSIGNED r, UNSIGNED n});
d t(ds L, ds R) := TRANSFORM
  SELF.r := L.r + R.r;
  SELF.n := L.n + R.n;
END;
ROLLUP(ds, t(LEFT, RIGHT), r);
/* results in:
  3  60
  3  40
  4  50
*/
ROLLUP(ds, LEFT.r = RIGHT.r, t(LEFT, RIGHT));
/* results in:
  2  30
  1  30
  3  40
  4  50
  the third record is not combined because the transform modified the value.
*/
```

Requerimentos da Função TRANSFORM - ROLLUP

Para as formas 1 e 2 de ROLLUP, a função *transform* deve adotar pelo menos dois parâmetros: um registro LEFT e um registro RIGHT, onde ambos devem estar no mesmo formato que o *recordset*. O formato do conjunto de registros resultante também deve ser o mesmo que as entradas.

Para a forma 3 de ROLLUP, a função *transform* deve adotar pelo menos dois parâmetros: um registro LEFT que deve estar no mesmo formato que o *recordset*, e um ROWS(LEFT) cujo formato deve ser um parâmetro DATASET(RECORDOF(*recordset*)). O formato do conjunto de registros resultante deve ser diferente das entradas.

Forma 1 do ROLLUP

A forma 1 é processada através de todos os registros no *recordset*, desempenhando a função *transform* apenas nos pares de registros adjacentes onde a *condition* de correspondência é atingida (indicando registros duplicados) e passando direto por todos os outros registros até o resultado.

Exemplo:

```
//a crosstab table of last names and the number of times they occur
MyRec := RECORD
  Person.per_last_name;
  INTEGER4 PersonCount := 1;
END;
LnameTable := TABLE(Person, MyRec); //create dataset to work with
SortedTable := SORT(LnameTable, per_last_name); //sort it first

MyRec Xform(MyRec L, MyRec R) := TRANSFORM
  SELF.PersonCount := L.PersonCount + 1;
  SELF := L; //keeping the L rec makes it KEEP(1), LEFT
// SELF := R; //keeping the R rec would make it KEEP(1), RIGHT
END;
XtabOut := ROLLUP(SortedTable,
  LEFT.per_last_name=RIGHT.per_last_name,
  Xform(LEFT, RIGHT));
```

Forma 2 do ROLLUP

A forma 2 é processada através de todos os registros no *recordset*, desempenhando a função *transform* apenas nos pares de registros adjacentes onde todas as expressões na *fieldlist* são correspondidas (indicando registros duplicados) e passando por todos os outros registros até o resultado. Esta forma permite usar o mesmo tipo de lógica de exclusão de campo EXCEPT disponível para DEDUP.

Exemplo:

```
rec := {STRING1 str1,STRING1 str2,STRING1 str3};
ds := DATASET([{'a', 'b', 'c'},{'a', 'b', 'c'},
               {'a', 'c', 'c'},{'a', 'c', 'd'}], rec);
rec tr(rec L, rec R) := TRANSFORM
    SELF := L;
END;
Cat(STRING1 L, STRING1 R) := L + R;
r1 := ROLLUP(ds, tr(LEFT, RIGHT), str1, str2);
    //equivalent to LEFT.str1 = RIGHT.str1 AND
    // LEFT.str2 = RIGHT.str2
r2 := ROLLUP(ds, tr(LEFT, RIGHT), WHOLE RECORD, EXCEPT str3);
    //equivalent to LEFT.str1 = RIGHT.str1 AND
    // LEFT.str2 = RIGHT.str2
r3 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD, EXCEPT str3);
    //equivalent to LEFT.str1 = RIGHT.str1 AND
    // LEFT.str2 = RIGHT.str2
r4 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD, EXCEPT str2,str3);
    //equivalent to LEFT.str1 = RIGHT.str1
r5 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD);
    //equivalent to LEFT.str1 = RIGHT.str1 AND
    // LEFT.str2 = RIGHT.str2 AND
    // LEFT.str3 = RIGHT.str3
r6 := ROLLUP(ds, tr(LEFT, RIGHT), str1 + str2);
    //equivalent to LEFT.str1+LEFT.str2 = RIGHT.str1+RIGHT.str2
r7 := ROLLUP(ds, tr(LEFT, RIGHT), Cat(str1,str2));
    //equivalent to Cat(LEFT.str1,LEFT.str2) =
    // Cat(RIGHT.str1,RIGHT.str2 )
```

Forma 3 do ROLLUP

A forma 3 é uma forma especial de ROLLUP onde o segundo parâmetro especificado para *transform* é um GROUP e o primeiro parâmetro é o primeiro registro nesse GROUP. Ela é processada através de todos os grupos no *recordset*, gerando um registro de resultado para cada grupo. Funções agregadas podem ser usadas dentro de *transform* (tais como TOPN ou CHOOSEN) no segundo parâmetro. O conjunto de registro de resultado não é agrupado. Esta forma é implicitamente LOCAL devido ao agrupamento.

Exemplo:

```
inrec := RECORD
    UNSIGNED6 did;
END;

outrec := RECORD(inrec)
    STRING20 name;
    UNSIGNED score;
END;

nameRec := RECORD
    STRING20 name;
END;

finalRec := RECORD(inrec)
```

```
    DATASET(nameRec) names;
    STRING20 secondName;
END;

ds := DATASET([1,2,3,4,5,6], inrec);

dsg := GROUP(ds, ROW);

i1 := DATASET([ {1, 'Kevin', 10},
                {2, 'Richard', 5},
                {5, 'Nigel', 2},
                {0, '', 0}], outrec);

i2 := DATASET([ {1, 'Kevin Halligan', 12},
                {2, 'Richard Charles', 15},
                {3, 'Blake Smith', 20},
                {5, 'Nigel Hicks', 100},
                {0, '', 0}], outrec);

i3 := DATASET([ {1, 'Halligan', 8},
                {2, 'Richard', 8},
                {6, 'Pete', 4},
                {6, 'Peter', 8},
                {6, 'Petie', 1},
                {0, '', 0}], outrec);

j1 := JOIN( dsg,
            i1,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER, MANY LOOKUP);

j2 := JOIN( dsg,
            i2,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER,
            MANY LOOKUP);

j3 := JOIN( dsg,
            i3,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER,
            MANY LOOKUP);

combined := REGROUP(j1, j2, j3);

finalRec doRollup(outRec l, DATASET(outRec) allRows) :=
    TRANSFORM
    SELF.did := l.did;
    SELF.names := PROJECT(allRows(score != 0),
                          TRANSFORM(nameRec, SELF := LEFT));
    SELF.secondName := allRows(score != 0)[2].name;
END;

results := ROLLUP(combined, GROUP, doRollup(LEFT, ROWS(LEFT)));
```

Ver também: Estrutura TRANSFORM, Estrutura RECORD, DEDUP, EXCEPT, GROUP

ROUND

ROUND(*realvalue*[, *decimals*])

<i>realvalue</i>	O valor de ponto flutuante a ser arredondado.
<i>decimals</i>	Opcional. Um valor inteiro que especifica o número de casas decimais que deve ser arredondado. Se omitida, o padrão é zero (resultado em valor inteiro).
Return:	ROUND retorna um único valor numérico.

A função **ROUND** retorna um *realvalue* arredondado usando o sistema de arredondamento aritmético padrão (casas decimais com valor inferior a ,5 são arredondadas para baixo e casas com valor acima ou igual a ,5 são arredondadas para cima).

Exemplo:

```
SomeRealValue1 := 3.14159;
INTEGER4 MyVal1 := ROUND(SomeRealValue1); // MyVal1 is 3
INTEGER4 MyVal2 := ROUND(SomeRealValue1,2); // MyVal2 is 3.14

SomeRealValue2 := 3.5;
INTEGER4 MyVal3 := ROUND(SomeRealValue2); // MyVal is 4

SomeRealValue3 := -1.3;
INTEGER4 MyVal4 := ROUND(SomeRealValue3); // MyVal is -1

SomeRealValue4 := -1.8;
INTEGER4 MyVal5 := ROUND(SomeRealValue4); // MyVal is -2
```

Ver também: ROUNDUP, TRUNCATE

ROUNDUP

ROUNDUP(*realvalue*)

<i>realvalue</i>	O valor de ponto flutuante a ser arredondado.
Return:	ROUNDUP retorna um único valor inteiro.

A função **ROUNDUP** retorna o valor inteiro arredondado do *realvalue* arredondando qualquer porção decimal para o próximo valor inteiro, independentemente do sinal.

Exemplo:

```
SomeRealValue := 3.14159;  
INTEGER4 MyVal := ROUNDUP(SomeRealValue); // MyVal is 4  
  
SomeRealValue := -3.9;  
INTEGER4 MyVal := ROUNDUP(SomeRealValue); // MyVal is -4
```

Ver também: ROUND, TRUNCATE

ROW

ROW({ *fields* } , *restruct*)

ROW(*row* , *resultrec*)

ROW([*row* ,] *transform*)

<i>fields</i>	Uma lista delimitada por vírgula dos valores de dados para cada campo no <i>restruct</i> contidos em chaves ({ }).
<i>restruct</i>	O nome da estrutura RECORD que define o layout do campo.
<i>row</i>	Uma linha única de dados. Pode ser um registro existente, ou valores de dados formatados em linha como a descrição do parâmetro dos campos acima, ou até mesmo um conjunto vazio ([]) para adicionar um registro limpo no formato do <i>resultrec</i> . Se omitida, o registro será gerado pela função TRANSFORM.
<i>resultrec</i>	Uma estrutura RECORD que define como criar uma linha de dados, semelhante ao tipo usado pela função TABLE.
<i>transform</i>	Uma função TRANSFORM que define como criar a linha de dados.
Return:	ROW retorna um único registro.

A função **ROW** cria um único registro de dados e é válida para uso em qualquer expressão onde um único registro seja válido.

Forma 1 do ROW

A primeira forma cria um registro a partir dos dados embutidos nos *campos*, estruturada como definido por *restruct*. É normalmente usada dentro de uma estrutura TRANSFORM como a expressão que define o resultado para um campo de dataset secundário.

Exemplo:

```
AkaRec := {STRING20 forename,STRING20 surname};
outputRec := RECORD
    UNSIGNED id;
    DATASET(AkaRec) kids;
END;
inputRec := {UNSIGNED id,STRING20 forename,STRING20 surname};
inPeople := DATASET([ {1,'Kevin','Halligan'}, {1,'Kevin','Hall'},
    {2,'Eliza','Hall'}, {2,'Beth','Took'} ],inputRec);
outputRec makeFatRecord(inputRec L) := TRANSFORM
    SELF.id := L.id;
    SELF.kids := DATASET([ { L.forename, L.surname } ],AkaRec);
END;
fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec L, outputRec R) := TRANSFORM
    SELF.id := L.id;
    SELF.kids := L.kids + ROW({R.kids[1].forename,R.kids[1].surname},AkaRec);
END;
r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
```

Forma 2 do ROW

A segunda forma cria um registro a partir da *row* especificada, usando *resultrec* da mesma maneira que a função TABLE opera. É normalmente usada dentro de uma estrutura TRANSFORM como a expressão que define o resultado para um campo de dataset secundário.

Exemplo:

```
AkaRec := {STRING20 forename,STRING20 surname};
outputRec := RECORD
  UNSIGNED id;
  DATASET(AkaRec) children;
END;
inputRec := {UNSIGNED id,STRING20 forename,STRING20 surname};
inPeople := DATASET([ {1,'Kevin','Halligan'}, {1,'Kevin','Hall'},
  {1,'Gawain',''}, {2,'Liz','Hall'},
  {2,'Eliza','Hall'}, {2,'Beth','Took'} ],inputRec);
outputRec makeFatRecord(inputRec L) := TRANSFORM
  SELF.id := L.id;
  SELF.children := ROW(L, AkaRec); //using Form 2 here
END;
fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec L, outputRec R) := TRANSFORM
  SELF.id := L.id;
  SELF.children := L.children +
    ROW({R.children[1].forename,R.children[1].surname},AkaRec);
END;
r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
```

Forma 3 do ROW

A terceira forma usa uma função TRANSFORM para gerar seu resultado de registro único. A função *transform* deve adotar pelo menos um parâmetro: um registro LEFT, o qual deve estar no mesmo formato que o registro de entrada. O formato do registro resultante deve ser distinto do de entrada.

Exemplo:

```
NameRec := RECORD
  STRING5 title;
  STRING20 fname;
  STRING20 mname;
  STRING20 lname;
  STRING5 name_suffix;
  STRING3 name_score;
END;

MyRecord := RECORD
  UNSIGNED id;
  STRING uncleanedName;
  NameRec Name;
END;

x := DATASET('RTTEST::RowFunctionData', MyRecord,THOR);

STRING73 CleanPerson73(STRING inputName) := FUNCTION
  suffix:=[ ' 0',' 1',' 2',' 3',' 4',' 5',' 6',' 7',' 8',' 9',
    ' J',' JR',' S',' SR'];
  InWords := Std.Str.CleanSpaces(inputName);
  HasSuffix := InWords[LENGTH(TRIM(InWords))-1 ..] IN suffix;
  WordCount := LENGTH(TRIM(InWords,LEFT,RIGHT)) - LENGTH(TRIM(InWords,ALL))+1;
  HasMiddle := WordCount = 5 OR (WordCount = 4 AND NOT HasSuffix) ;
  Space1 := Std.Str.Find(InWords,' ',1);
  Space2 := Std.Str.Find(InWords,' ',2);
  Space3 := Std.Str.Find(InWords,' ',3);
  Space4 := Std.Str.Find(InWords,' ',4);
  STRING5 title := InWords[1..Space1-1];
  STRING20 fname := InWords[Space1+1..Space2-1];
```

```
STRING20 mname := IF(HasMiddle, InWords[Space2+1..Space3-1], '');
STRING20 lname := MAP(HasMiddle AND NOT HasSuffix =>
    InWords[Space3+1..],
    HasMiddle AND HasSuffix =>
    InWords[Space3+1..Space4-1],
    NOT HasMiddle AND NOT HasSuffix =>
    InWords[Space2+1..],
    NOT HasMiddle AND HasSuffix =>
    InWords[Space2+1..Space3-1],
    '');
STRING5 name_suffix := IF(HasSuffix, InWords[LENGTH(TRIM(InWords))-1 ..], '');
STRING3 name_score := '';
RETURN title + fname + mname + lname + name_suffix + name_score;
END;

//Example 1 - a transform to create a row from an uncleaned name
NameRec createRow(string inputName) := TRANSFORM
    cleanedText := CleanPerson73(inputName);
    SELF.title := cleanedText[1..5];
    SELF.fname := cleanedText[6..25];
    SELF.mname := cleanedText[26..45];
    SELF.lname := cleanedText[46..65];
    SELF.name_suffix := cleanedText[66..70];
    SELF.name_score := cleanedText[71..73];
END;

myRecord t(myRecord L) := TRANSFORM
    SELF.Name := ROW(createRow(L.uncleanedName));
    SELF := L;
END;
y := PROJECT(x, t(LEFT));
OUTPUT(y);

//Example 2 - an attribute using that transform to generate the row.
NameRec cleanedName(STRING inputName) := ROW(createRow(inputName));
myRecord t2(myRecord L) := TRANSFORM
    SELF.Name := cleanedName(L.uncleanedName);
    SELF := L;
END;
y2 := PROJECT(x, t2(LEFT));
OUTPUT(y2);

//Example 3 = Encapsulate the transform inside the attribute by
// defining a FUNCTION structure.
NameRec cleanedName2(STRING inputName) := FUNCTION

    NameRec createRow := TRANSFORM
        cleanedText := CleanPerson73(inputName);
        SELF.title := cleanedText[1..5];
        SELF.fname := cleanedText[6..25];
        SELF.mname := cleanedText[26..45];
        SELF.lname := cleanedText[46..65];
        SELF.name_suffix := cleanedText[66..70];
        SELF.name_score := cleanedText[71..73];
    END;

    RETURN ROW(createRow); //omitted row parameter
END;

myRecord t3(myRecord L) := TRANSFORM
    SELF.Name := cleanedName2(L.uncleanedName);
    SELF := L;
END;
```

```
y3 := PROJECT(x, t3(LEFT));  
OUTPUT(y3);
```

Ver também: Estrutura TRANSFORM, DATASET, Estrutura RECORD, Estrutura FUNCTION

ROWDIFF

ROWDIFF(*left*, *right* [, **COUNT**])

<i>left</i>	O registro left, ou uma estrutura de registro aninhado.
<i>RIGHT</i>	O registro right, ou uma estrutura de registro aninhado.
COUNT	Opcional. Especifica o retorno de um conjunto delimitado por vírgula de zeros e um (0,1), indicando quais campos coincidiram (0) e quais não (1). Se omitido, retornará um conjunto delimitado por vírgula dos nomes de campo não correspondentes.
Return:	ROWDIFF retorna um único valor.

A função **ROWDIFF** é válida somente para uso dentro da estrutura TRANSFORM de uma operação JOIN, e é usada como a expressão que define o resultado de um campo da string. Os campos são correspondidos por nome e apenas os campos de mesmo nome são incluídos no resultado.

Exemplo:

```
FullName := RECORD
  STRING30 forename;
  STRING20 surname;
  IFBLOCK(SELF.surname <> 'Windsor')
    STRING20 middle;
  END;
END;
in1rec := {UNSIGNED1 id,FullName name,UNSIGNED1 age,STRING5 title};
in2rec := {UNSIGNED1 id,FullName name,REAL4 age,BOOLEAN dead};
in1 := DATASET([
  {1,'Kevin','Halligan','',33,'Mr'},
  {2,'Liz','Halligan','',33,'Dr'},
  {3,'Elizabeth','Windsor',99,'Queen'}], in1rec);
in2 := DATASET([
  {1,'Kevin','Halligan','',33,false},
  {2,'Liz','', 'Jean',33,false},
  {3,'Elizabeth','Windsor',99.1,false}], in2rec);
outrec := RECORD
  UNSIGNED1 id;
  STRING35 diff1;
  STRING35 diff2;
  STRING35 diff3;
  STRING35 diff4;
END;
outrec t1(in1 L, in2 R) := TRANSFORM
  SELF.id := L.id;
  SELF.diff1 := ROWDIFF(L,R);
  SELF.diff2 := ROWDIFF(L.name, R.name);
  SELF.diff3 := ROWDIFF(L, R, COUNT);
  SELF.diff4 := ROWDIFF(L.name, R.name, COUNT);
END;
OUTPUT(JOIN(in1, in2, LEFT.id = RIGHT.id, t1(LEFT,RIGHT)));
// The result set from this code is:
//id diff1 diff2 diff3 diff4
//1 0,0,0,0,0 0,0,0
//2 name.surname,name.middle surname,middle 0,0,1,1,0 0,1,1
//3 age 0,0,0,0,1 0,0,0
```

Ver também: Estrutura TRANSFORM, JOIN

SAMPLE

SAMPLE(*recordset*, *interval* [, *which*] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para amostragem. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
<i>interval</i>	O intervalo entre os registros a serem retornados.
<i>which</i>	Opcional. Um número inteiro que especifica o número ordinal do conjunto de amostra a ser retornado. Isso é usado para obter múltiplas amostras sem sobreposição a partir de um mesmo conjunto de registros.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	SAMPLE retorna um conjunto de registros.

A função **SAMPLE** retorna uma amostra de conjunto de registros a partir do *recordset* nominado.

Exemplo:

```
MySample := SAMPLE(Person,10,1) // get every 10th record

SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                    {'F'},{'G'},{'H'},{'I'},{'J'},
                    {'K'},{'L'},{'M'},{'N'},{'O'},
                    {'P'},{'Q'},{'R'},{'S'},{'T'},
                    {'U'},{'V'},{'W'},{'X'},{'Y'}],
                    {STRING1 Letter});
Set1 := SAMPLE(SomeFile,5,1); // returns A, F, K, P, U
```

Ver também: **CHOOSE**, **ENTH**

SEQUENTIAL

[*definitionname* :=] **SEQUENTIAL**(*actionlist*)

<i>definitionname</i>	Opcional. O nome da ação, que transforma a ação em uma definição, consequentemente não é executado até que <i>definitionname</i> seja usado como uma ação.
<i>actionlist</i>	Uma lista delimitada por vírgula das ações a serem executadas na ordem. Podem ser ações ECL ou ações externas.

A ação **SEQUENTIAL** executa os itens da *actionlist* na ordem em que aparecem na *actionlist*.

Exemplo:

```
Act1 := OUTPUT(A_People,OutputFormat1,'//hold01/fred.out');
Act2 := OUTPUT(Person,{Person.per_first_name,Person.per_last_name});
Act3 := OUTPUT(Person,{Person.per_last_name});
//by naming these actions, they become inactive definitions
//that only execute when the definition names are called as definitions

SEQUENTIAL(Act1,PARALLEL(Act2,Act3));
//executes Act1 alone, and only when it's finished,
//executes Act2 and Act3 together
```

Ver também: ORDERED, PARALLEL,PERSIST

SET

SET(*recordset*, *field* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros do qual o SET de valores será derivado.
<i>field</i>	O campo no recordset do qual serão obtidos os valores.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for "False" (Falso), especifica que a ordem do registro de resultado não é importante. Quando for "True" (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	SET retorna um SET de valores do mesmo tipo do campo.

A função **SET** retorna um SET para ser usado em qualquer operação de conjunto (como o operador IN), semelhante a uma seleção secundária no SQL quando usada com o operador IN. Ela não remove elementos em duplicidade e não ordena o conjunto.

Um problema comum é o uso da função SET em uma condição de filtro. Por exemplo:

```
MyDS := myDataset(myField IN SET(anotherDataset, someField));
```

O código gerado para isso é ineficiente se "anotherDataset" contiver um grande número de elementos, podendo também causar o erro no qual o dataset é muito grande para gerar o resultado da tarefa ("Dataset too large to output to workunit"). Uma maneira mais eficiente de regravar a expressão seria:

```
MyDS := JOIN(myDataset, anotherDataset, LEFT.myField = RIGHT.someField, TRANSFORM(LEFT), LOOKUP) ;
```

O resultado final é o mesmo, o conjunto de registros "myDataset" no qual o valor "myField" é um dos valores "someField" de "anotherDataset", mas o código possui uma execução bem mais eficiente.

Você pode construir um DATASET a partir de um SET.

```
ds := DATASET([{'X',1},{ 'B',3},{ 'C',2},{ 'B',5},
               {'C',4},{ 'D',6},{ 'E',2}],
              {STRING1 Ltr, INTEGER1 Val});
s1 := SET(ds,Ltr);           //a SET of just the Ltr field values:
DATASET(s1,{STRING1 Ltr});  //a DATASET from the SET
```

Exemplo:

```
ds := DATASET([{'X',1},{ 'B',3},{ 'C',2},{ 'B',5},
               {'C',4},{ 'D',6},{ 'E',2}],
              {STRING1 Ltr, INTEGER1 Val});

//a SET of just the Ltr field values:
s1 := SET(ds,Ltr);
```

```
COUNT(s1); //results in 7
s1;        //results in ['X','B','C','B','C','D','E']

//a simple way to get just the unique elements
//is to use a crosstab TABLE:
t := TABLE(ds,{Ltr},Ltr); //order indeterminant

s2 := SET(t,Ltr);
COUNT(s2); //results in 5
s2;        //results in  ['D','X','C','E','B']

//sorted unique elements
s3 := SET(SORT(t,Ltr),Ltr);
COUNT(s3); //results in 5
s3;        //results in ['B','C','D','E','X']
```

Ver também: Conjuntos e Filtros, SET OF, Operadores Set, Operador IN

SIN

SIN(*angle*)

<i>angle</i>	O valor radiano REAL para o qual será localizado o seno.
Return:	SIN retorna um valor REAL único.

A função **SIN** retorna o seno do *ângulo*.

Exemplo:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
Sine45  := SIN(Angle45);    //get sine of the 45 degree angle
```

Ver também: ACOS, COS, ASIN, TAN, ATAN, COSH, SINH, TANH

SINH

SINH(*angle*)

<i>angle</i>	O valor radiano REAL para o qual será localizado o seno hiperbólico.
Return:	SINH retorna um valor REAL único.

A função **SINH** retorna o seno hiperbólico do *ângulo*.

Exemplo:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
HyperbolicSine45 := SINH(Angle45); //get hyperbolic sine of the angle
```

Ver também: ACOS, COS, ASIN, TAN, ATAN, COSH, SIN, TANH

SIZEOF

SIZEOF(*data* [, **MAX**])

<i>data</i>	O nome de um dataset, uma estrutura RECORD, o nome de um campo totalmente qualificado, ou uma expressão de string constante.
MAX	Especifica que os dados são de comprimento variável (que contêm datasets secundários) e que o valor a ser retornado possui tamanho máximo.
Return:	ROUNDUP retorna um único valor inteiro.

A função **SIZEOF** retorna o número total de bytes definido para armazenar a estrutura de dados ou campo especificados. *data* Estrutura ou Campo.

Exemplo:

```
MyRec := RECORD
  INTEGER1 F1;
  INTEGER5 F2;
  STRING1 F3;
  STRING10 F4;
  QSTRING12 F5;
  VARSTRING12 F6;
END;
MyData :=
  DATASET([ {1,333333333333,'A','A','A','V'A'} ],MyRec);
SIZEOF(MyRec); //result is 39
SIZEOF(MyData.F1); //result is 1
SIZEOF(MyData.F2); //result is 5
SIZEOF(MyData.F3); //result is 1
SIZEOF(MyData.F4); //result is 10
SIZEOF(MyData.F5); //result is 9 -12 chars stored in 9
                    bytes
SIZEOF(MyData.F6); //result is 13 -12 chars plus null
                    terminator

Layout_People := RECORD
  STRING15 first_name;
  STRING15 middle_name;
  STRING25 last_name;
  STRING2 suffix;
  STRING42 street;
  STRING20 city;
  STRING2 st;
  STRING5 zip;
  STRING1 sex;
  STRING3 age;
  STRING8 dob;
  BOOLEAN age_flag;
  UNSIGNED8 __filepos {VIRTUAL(fileposition)};
END;
File_People := DATASET('ecl_training::People', Layout_People,
  FLAT);
SIZEOF(File_People); //result is 147
SIZEOF(File_People.street); //result is 42
SIZEOF('abc' + '123'); //result is 6
SIZEOF(person.per_cid); //result is 9 - Person.per_cid is
  DATA9
```

Ver também: **LENGTH**

SOAPCALL

result := SOAPCALL([*reset*,] *url*, *service*, *instructure*, [*transform*,] DATASET(*outstructure*) / *outstructure* [*options*] [, UNORDERED | ORDERED(*bool*)] [, STABLE | UNSTABLE] [, PARALLEL [(*numthreads*)]] [, ALGORITHM(*name*)]);

SOAPCALL([*reset*,] *url*, *service*, *instructure*, [*transform*,] [*options*] [, UNORDERED | ORDERED(*bool*)] [, STABLE | UNSTABLE] [, PARALLEL [(*numthreads*)]] [, ALGORITHM(*name*)]);

<i>result</i>	Nome de atributo para o conjunto de registro resultante ou registro individual.
<i>reset</i>	Opcional. O recordset da entrada. Se omitida, um registro individual de entrada deve ser definido por valores padrão para cada campo do parâmetro <i>instructure</i> .
<i>url</i>	<p>Uma string que contém uma lista de URLs delimitados por pipe () que hospedam o serviço a ser invocado (podem anexar nomes de módulos de repositório). Isso objetiva fornecer meios para o client realizar uma busca federada onde a solicitação é enviada para cada um dos sistemas de destino na lista. Estes URLs podem conter nomes de usuário e senhas em formato padrão, se exigidos. Os nomes de usuário/senhas padrão são aqueles contidos na tarefa. Se acionar um serviços Web ESP, é possível anexar o parâmetro <i>ver_=n.nn</i> para especificar a versão do serviço. Por exemplo:</p> <pre>SOAPCALL('http://127.0.0.1:8010/Wsdfu/?ver_=1.22', 'DFUSearchData', instructure, DATASET(outstructure));</pre>
<i>service</i>	Uma expressão da string que contém o nome do serviço a ser invocado. Isso pode estar na forma <i>module.attribute</i> se o serviço estiver em uma plataforma Roxie.
<i>instructure</i>	Uma estrutura RECORD que contém as definições do campo de entrada a partir das quais a entrada XML é construída para o serviço SOAP. O nome das tags no XML são derivados dos nomes em minúsculas dos campos no registro de entrada; isso pode ser substituído colocando um xpath no campo ({xpath('tagname')}) - veja a seção XPATH Support da discussão RECORD Structure). Se o parâmetro <i>reset</i> não estiver presente, cada definição de campo deve conter um valor padrão que constituirá o registro de entrada individual. Se o parâmetro <i>reset</i> estiver presente, cada definição de campo deve conter um valor padrão, a menos que transform também esteja especificada para fornecer valores de dados.
<i>transform</i>	Opcional. A função TRANSFORM é usada para acionar o processamento dos dados <i>instructure</i> . Isso elimina a necessidade de definir valores padrão para todos os campos na estrutura RECORD <i>instructure</i> . A função deve adotar pelo menos um parâmetro: um registro LEFT de mesmo formato que a entrada <i>reset</i> . O formato do conjunto de registro resultante deve ser o mesmo que a entrada <i>instructure</i> .
DATASET(<i>outstructure</i>)	Especifica o resultado do <i>recordset</i> no formato <i>outstructure</i> .
<i>outstructure</i>	Uma estrutura RECORD que contém as definições do campo de resultado. Se não for usada como parâmetro para a palavra-chave DATASET, ela especifica um resultado de registro individual. Cada definição de campo na estrutura RECORD deve usar um atributo xpath ({xpath('tagname')}) para eliminar problemas de distinção entre maiúsculas e minúsculas.
<i>options</i>	Uma lista delimitada por vírgula das especificações opcionais da lista abaixo.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.

Referência a Linguagem ECL
Ações e Funções Built-in

<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	SOAPCALL retorna um conjunto de registros, um registro individual ou nenhum.

SOAPCALL são funções ou ações que acionam um serviço SOAP (Protocolo Simples de Acesso a Objetos).

Opções válidas são:

RETRY (<i>count</i>)	Especifica quantas vezes houve uma nova tentativa de acionamentos, se ocorrerem erros não fatais. Se omitida, o padrão é três (3).
TIMEOUT (<i>period</i>)	Especifica o número tentativas de leitura antes da falha. O <i>period</i> (período) é um número real cuja parte inteira especifica os segundos. Definir para zero (0) indica espera permanente. Se omitido, o padrão é trezentos (300).
TIMELIMIT (<i>period</i>)	Especifica o tempo total permitido para o SOAPCALL. O <i>period</i> (período) é um número real cuja parte inteira especifica os segundos. Se omitido, o padrão é zero (0) – indicando ausência de limite de tempo.
HEADING (<i>prefix,suffix</i>)	Especifica as tags para encapsular os campos de entrada XML . Se omitido, o padrão é: HEADING("","").
XPATH (<i>xpath</i>)	Especifica o caminho usado para acessar as linhas no resultado. Se omitido, o padrão é: 'serviceResponse/Results/Result/Dataset/Row'.
MERGE (<i>n</i>)	Especifica o processamento de <i>n</i> registros por lote (o bloqueio). Se omitido, o padrão é 1 (valores diferentes de 1 podem ser incompatíveis com serviços não Roxie). Válido para ser usado apenas se o parâmetro <i>reset</i> também estiver presente.
PARALLEL (<i>n</i>)	Especifica o número de threads simultâneas – para processar as consultas do Motor de entrega de dados – para no máximo 50 (o padrão é 2). Isso busca limitar o número de sessões simultâneas.
ONFAIL (<i>transform</i>)	Especifica acionar a função transform, se o serviço falhar em relação a um registro específico, ou a palavra-chave SKIP. A função TRANSFORM deve produzir um <i>resultype</i> igual à <i>outstructure</i> e pode usar o FAILCODE e/ou FAILMESSAGE para fornecer detalhes da falha.
TRIM	Especifica que todos os espaços à direita são removidos das strings antes do resultado.
RESPONSE (<i>NOTRIM</i>)	Define o indicador para impedir a remoção de espaços na resposta.
NAMESPACE (<i>namespace</i>)	Especifica o nível superior do <i>NAMESPACE</i> para uma chamada SOAP
LITERAL	Especifica que o serviço não é necessariamente implementado no ESP.
SOAPACTION (<i>value</i>)	Especifica um <i>valor</i> onde esse <i>valor</i> é uma expressão da string que normalmente contém um URN ou URL mandatário para uma interoperabilidade adequada do <i>serviço</i> Web.

LOG	Se especificado, grava detalhes no arquivo de log do motor (hThor, Thor, ou Roxie) para o qual SOAPCALL é enviado.
LOG (MIN)	Especifica gravar detalhes mínimos do SOAPCALL em um arquivo de log.
LOG (expression)	Especifica adicionar a expressão no log ao desempenhar um SOAPCALL.
ENCODING	Especifica que o serviço Web acionado exige um formato de mensagem diferente, onde o tipo de informação esteja incorporado em XML.
HTTPHEADER	Especifica as informações do cabeçalho a serem passadas para o serviço. SOAPCALL suporta múltiplas instâncias da opção HTTPHEADER se você precisa especificar várias strings de chave/valor de cabeçalho.

Função SOAPCALL

Esta forma de função SOAPCALL pode adotar como entrada tanto um registro individual quanto um conjunto de registros, e ambos os tipos de registros podem resultar em um registro individual ou em um conjunto de registros.

A definição do registro de resultado *outstructure* pode conter um campo inteiro com um XPATH de "_call_latency" para receber o tempo (em segundos) do acionamento que gerou a linha (desde a criação do soquete ao recebimento da resposta). A latência é inserida em cada linha de retorno do acionamento. Assim, se um acionamento demorou 90 segundos e retornou 11 linhas, você então verá 11 linhas com 90 no campo _call_latency.

Exemplo:

```
OutRec1 := RECORD
  STRING500 OutData{XPATH('OutData')};
  UNSIGNED4 Latency{XPATH('_call_latency')};
END;

ip := 'http://127.0.0.1:8022/';
ips := 'https://127.0.0.1:8022/';
ipspw := 'https://username:password@127.0.0.1:8022/';
svc := 'MyModule.SomeService';

//1 rec in, 1 rec out
OneRec1 := SOAPCALL(ips,svc,{STRING500 InData := 'Some Input Data'},OutRec1);

//1 rec in, recordset out
ManyRec1 := SOAPCALL(ip,svc,{STRING500 InData := 'Some Input Data'},DATASET(OutRec1));

//recordset in, 1 rec out
OneRec2 := SOAPCALL(InputDataset,ip,svc,{STRING500 InData},OutRec1);

//recordset in, recordset out
ManyRec2 :=
  SOAPCALL(InputDataset,ipspw,svc,{STRING500 InData := 'Some In Data'},DATASET(OutRec1));

//TRANSFORM function usage example
namesRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  INTEGER2 age := 25;
END;
ds := DATASET('x',namesRecord,FLAT);

inRecord := RECORD
  STRING name{xpath('Name')};
  UNSIGNED6 id{XPATH('ADL')};
END;
outRecord := RECORD
  STRING name{xpath('Name')};
  UNSIGNED6 id{XPATH('ADL')};
```

```
    REAL8 score;
END;
inRecord t(namesRecord l) := TRANSFORM
    SELF.name := l.surname;
    SELF.id := l.age;
END;
outRecord genDefault1() := TRANSFORM
    SELF.name := FAILMESSAGE;
    SELF.id := FAILCODE;
    SELF.score := (REAL8)FAILMESSAGE('ip');
END;
outRecord genDefault2(namesRecord l) := TRANSFORM
    SELF.name := l.surname;
    SELF.id := l.age;
    SELF.score := 0;
END;

ip := 'http://127.0.0.1:8022/';
svc:= 'MyModule.SomeService';
OUTPUT(SOAPCALL(ip, svc, { STRING20 surname := 'Halligan', STRING20 forename := 'Kevin'; },
DATASET(outRecord), ONFAIL(genDefault1())));

OUTPUT(SOAPCALL(ds, ip, svc, inRecord, t(LEFT), DATASET(outRecord),
    ONFAIL(genDefault2(LEFT))));

OUTPUT(SOAPCALL(ds, ip, svc, inRecord, t(LEFT), DATASET(outRecord), ONFAIL(SKIP)));

//Using HTTPHEADER to pass Authorization info
OUTPUT(SOAPCALL(ds, ip, svc, inRecord, t(LEFT), DATASET(outRecord), ONFAIL(SKIP),
    HTTPHEADER('Authorization', 'Basic dXNlcm5hbWU6cGFzc3dvcmQ=')));
```

Ação SOAPCALL

A segunda forma de SOAPCALL (a ação), pode adotar como entrada um registro individual ou um conjunto de registros. Nenhum dos tipos de entrada gera qualquer resultado retornado – ele simplesmente inicia o serviço SOAP especificado, fornecendo os dados de entrada.

Exemplo:

```
//1 rec in, no result
SOAPCALL( 'https://127.0.0.1:8022/', 'MyModule.SomeService',
    {STRING500 InData := 'Some Input Data'});

//recordset in, no result
SOAPCALL( InputDataset, 'https://127.0.0.1:8022/', 'MyModule.SomeService', {STRING500 InData});
```

Ver também: Estrutura RECORD, Estrutura TRANSFORM

SORT

SORT(*recordset*,*value* [**JOINED**(*joinedset*)][**SKEW**(*limit* [,*target*])][**THRESHOLD**(*size*)][**LOCAL**][**FEW**][**STABLE** [(*algorithm*)] | **UNSTABLE** [(*algorithm*)]] [**UNORDERED** | **ORDERED**(*bool*)] [**PARALLEL** [(*numthreads*)]] [**ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um record set derivado de algumas condições de filtro, ou qualquer expressão que resulte em um record set derivado.
<i>value</i>	Uma lista delimitada por vírgula das expressões ou dos campos principais no recordset em que ocorrerá a classificação, sendo leftmost o critério de classificação mais significativo. Um sinal de subtração a frente (-) indica uma classificação em ordem decrescente daquele elemento. Você pode ter vários parâmetros de valor para indicar os tipos na classificação. É possível usar as palavras-chave RECORD (ou WHOLE RECORD) para indicar uma classificação na ordem crescente em todos os campos, e/ou usar a palavra-chave EXCEPT para listar os campos que não serão classificados no recordset.
JOINED	Opcional. Indica que esta classificação usará os mesmos pontos-base já usados por <i>joinedset</i> , para que os recursos correspondentes entre o recordset e <i>joinedset</i> fiquem nos mesmos nós do supercomputador. Usado para otimizar as junções do supercomputador onde <i>joinedset</i> é muito grande e recordset é pequeno.
<i>joinedset</i>	Um conjunto de registros que já foi classificado anteriormente pelos mesmos parâmetros de valor do recordset.
SKEW	Opcional. Indica que você sabe que os dados não são espalhados uniformemente entre os nós (são distorcidos) e você opta por substituir o padrão especificando seu próprio valor limite para permitir que a tarefa continue apesar da distorção.
<i>limit</i>	Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima da distorção a ser permitida antes que a tarefa falhe (o padrão é 0,1 = 10%).
<i>target</i>	Opcional. Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima desejada da distorção a ser permitida (o padrão é 0,1 = 10%).
THRESHOLD	Opcional. Indica o tamanho mínimo de uma única parte do recordset antes que o limite SKEW seja aplicado.
<i>size</i>	Um valor inteiro indicando o número mínimo de bytes para uma parte única.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior. Ocorrerá um erro se o recordset tiver sido GROUPed.
FEW	Opcional. Especifica que poucos registros serão classificados. Isso evita lançar SORT no disco se outra atividade intensiva de recursos estiver sendo executada simultaneamente.
STABLE	Opcional. Especifica uma classificação estável – as duplicações são exibidas na mesma ordem da entrada. Este é o padrão se nenhuma classificação STABLE ou UNSTABLE for especificada. É ignorado caso não seja suportado pela plataforma de destino.
<i>algorithm</i>	Opcional. Uma constante da string que especifica o algoritmo de classificação a ser usado (consulte a lista de valores válidos abaixo). Se omitida, o algoritmo padrão depende da plataforma que será o alvo da consulta.
UNSTABLE	Opcional. Especifica uma classificação instável – as duplicações podem ser exibidas em qualquer ordem. É ignorado caso não seja suportado pela plataforma de destino.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.

ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	SORT retorna um conjunto de registros.

A função **SORT** ordena o *recordset* de acordo com os *valores* especificados e (se LOCAL não for especificado) divide o resultado de forma que todos os registros com mesmos *valores* estejam no mesmo nó. SORT é geralmente usada para gerar conjuntos de registro operados pelas funções DEDUP, GROUP, e , para que essas funções possam operar de forma ideal. A classificação do resultado final é, obviamente, outro método comum de uso.

Algoritmos de classificação

Existem três algoritmos de classificação disponíveis: quicksort insertionsort heapsort. Os três não estão disponíveis em todas as plataformas. A especificação de um algoritmo inválido para a plataforma visada gerará um aviso, e o algoritmo padrão dessa plataforma será implementado.

Thor	Suporta quicksort estável e instável – se necessário, a classificação será lançada no disco. A classificação paralela ocorre automaticamente em clusters com nós de várias CPUs ou núcleo de várias CPUs.
hthor	Suporta quicksort estável e instável, insertionsort estável e instável, e heapsort estável – a classificação será lançada no disco se necessário. Heapsort estável será padrão se ambos STABLE e UNSTABLE forem omitidos ou se STABLE estiver presente sem um parâmetro de algoritmo.
	Quicksort instável será padrão se UNSTABLE estiver presente sem um parâmetro de algoritmo.
Roxie	Suporta quicksort instável, insertionsort estável, e heapsort estável – a classificação não é lançada no disco.
	Heapsort estável será padrão se ambos STABLE e UNSTABLE forem omitidos ou se STABLE estiver presente sem um parâmetro de algoritmo. Quando há mais de 1024 linhas, insertionsort implementa blocking e heapmerging.

Classificação rápida

Uma classificação rápida não faz nada até receber a última linha de sua entrada, não produzindo nenhum resultado até a classificação ser concluída, de forma que o tempo necessário para realizar a classificação não pode se sobrepor ao tempo para processar a entrada nem para produzir seu resultado. Em circunstâncias normais, esse tipo de classificação leva o menor tempo possível da CPU. Há casos excepcionais e raros onde ela pode ser realizada de forma inadequada (o famoso "problema de mediana de três" é um exemplo), mas é improvável que você cruze com um desses no caminho.

Em um cluster Thor onde cada nó possui múltiplas CPUs ou núcleos de CPU, é possível dividir o problema de classificação rápida e executar seções do trabalho em paralelo. Isso acontece automaticamente se o hardware oferecer suporte para o recurso. Fazer isso não melhora a quantidade de tempo real de CPU usado (na verdade, em um ponto de vista fracionado, ele até aumenta devido à sobrecarga de divisão da tarefa), mas o tempo geral necessário para realizar

a operação de classificação é significativamente reduzido. Em um cluster com duas CPUs/nós de núcleo, deve levar apenas metade do tempo, cerca de um quarto do tempo em um cluster com nós de processador de quatro núcleos e assim por diante.

Classificação por Inserção

Uma classificação de inserção faz todo esse trabalho enquanto recebe sua entrada. Observe que o algoritmo usado realiza uma pesquisa binária para inserção (diferentemente de uma classificação de inserção clássica). Em circunstâncias normais, essa classificação deve gerar o pior tempo de CPU. Em casos nos quais a fonte de entrada seja lenta, mas não vinculada à CPU (por exemplo, uma leitura de dados remota e lenta ou entrada de um SOAPCALL lento), o tempo necessário para realizar a classificação é completamente sobreposto à entrada.

Classificação em Pilha

Uma classificação heap realiza cerca de metade de seu trabalho ao receber a entrada e a outra metade ao gerar o resultado. Em circunstâncias normais, espera-se que leve mais tempo de CPU do que uma classificação rápida, mas menos do que uma classificação de inserção. Dessa forma, em consultas nas quais a fonte de entrada é lenta, mas não vinculada à CPU, metade do tempo necessário para realizar a classificação é sobreposto à entrada. Da mesma maneira, em consultas nas quais o processamento do resultado é lento, mas não vinculado à CPU, a outra metade do tempo necessário para realizar a classificação é sobreposta ao resultado. Além disso, se o processamento de classificação terminar sem consumir toda a sua entrada, então parte do seu trabalho pode ser totalmente evitada (cerca de metade, no caso de limite no qual não há consumo do resultado), economizando tempo de CPU e tempo total.

Em alguns casos, como quando uma SORT é rapidamente seguida por um CHOOSEN, o compilador poderá detectar que apenas uma parte do resultado de classificação será necessária e substituí-la por uma implementação mais eficiente. Isso não será aplicável no caso geral.

Estável vs. Instável

Uma classificação estável é necessária quando a entrada possa conter itens em duplicidade (isso é, registros que possuem os mesmos valores para todos os campos de classificação) e será necessário que os itens em duplicidade apareçam no resultado na mesma ordem que aparecem na entrada. Quando a entrada não contém itens em duplicidade, ou quando você não se importa em qual ordem os itens em duplicidade aparecem no resultado, uma classificação instável será suficiente.

Normalmente, uma classificação instável será um pouco mais rápida do que a versão estável do mesmo algoritmo. No entanto, em casos nos quais o algoritmo de classificação ideal só está disponível em uma versão estável, ele pode, muitas vezes, ser melhor do que a versão instável de um algoritmo diferente.

Considerações sobre desempenho

A discussão a seguir se aplica principalmente às classificações locais, já que o Thor é a única plataforma que realiza classificações globais e não proporciona uma escolha de algoritmos.

Tempo de CPU vs. tempo total

Em algumas situações, uma consulta pode levar o menor tempo de CPU usando uma classificação rápida, mas pode levar mais tempo total porque o tempo de classificação não pode ser sobreposto ao tempo exigido por uma tarefa com alto consumo de E/S antes ou depois dela. Em um sistema no qual apenas um subgrafo ou consulta é executado por vez (THor ou hthor), isso pode fazer da classificação rápida uma escolha inadequada – já que o tempo extra é simplesmente desperdiçado. Em um sistema no qual vários subgrafos ou consultas são executados simultaneamente (como um Roxie carregado) há uma concessão, já que minimizar o tempo total reduzirá a latência da consulta específica – mas minimizar o tempo de CPU maximizará o rendimento de todo o sistema.

Ao considerar a classificação rápida paralela, podemos ver que ela deve reduzir significativamente a latência para essa consulta; mas, se outras CPUs/núcleos estiverem em uso para outras tarefas (como quando dois Thors estão em execução nas mesmas duas máquinas de núcleo/CPUs), isso não vai aumentar (e diminuirá levemente) o rendimento das máquinas.

Gravando em disco

Normalmente, os registros são classificados na memória. Quando não há memória suficiente, pode ocorrer vazamento para o disco. Isso significa que blocos de registros são classificados na memória e gravados em disco e os blocos classificados são então combinados do disco na conclusão. Isso diminui expressivamente a velocidade da classificação. Isso também significa que o tempo de processamento para a classificação de heap será mais longo, já que não poderá mais se sobrepor ao seu resultado.

Quando não houver memória suficiente para manter todos os registros e o vazamento para disco não estiver disponível (como na plataforma Roxie), a consulta falhará.

Como a classificação afeta JOINS

Uma operação normal de JOIN exige que ambas as suas entradas sejam classificadas pelos campos usados nas mesmas proporções da condição de correspondência. O supercomputador realiza automaticamente essas classificações "nos bastidores", exceto se souber que uma entrada já tenha sido classificada corretamente. Dessa forma, algumas das considerações que se aplicam na hora de levar em conta um algoritmo para um SORT também podem ser aplicáveis a um JOIN. Para aproveitar esses algoritmos de classificação alternativos em um contexto de JOIN, é necessário usar SORT nos datasets de entrada como desejado e depois especificar a opção NOSORT no JOIN.

Observe que não é necessário realizar classificação para operações de JOIN usando as opções KEYED (ou meia chave), LOOKUP ou ALL. Em algumas circunstâncias (normalmente em consultas Roxie ou em casos nos quais o otimizador acredita que há poucos registros no dataset de entrada direito), o otimizador do supercomputador realizará automaticamente uma LOOKUP ou JOIN ALL em vez de JOIN regular. Isso significa que, se você tiver realizado SORT e especificado a opção NOSORT no JOIN, acabará perdendo essa possível otimização.

Exemplo:

```
MySet1 := SORT(Person,-last_name, first_name);  
// descending last name, ascending first name  
  
MySet2 := SORT(Person,RECORD,EXCEPT per_sex,per_marital_status);  
// sort by all fields except sex and marital status  
  
MySet3 := SORT(Person,last_name, first_name,STABLE('quicksort'));  
// stable quick sort, not supported by Roxie  
  
MySet4 := SORT(Person,last_name, first_name,UNSTABLE('heapsort'));  
// unstable heap sort,  
// not supported by any platform,  
// therefore ignored  
  
MySet5 := SORT(Person,last_name,first_name,STABLE('insertionsort'));  
// stable insertion sort, not supported by Thor
```

Ver também: SORTED, RANK, RANKED, EXCEPT

SORTED

SORTED(*recordset,value*)

SORTED(*index*)

<i>recordset</i>	O conjunto de registros classificados. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
<i>value</i>	Uma lista delimitada por vírgula das expressões ou dos campos principais nos quais ele foi classificado, sendo leftmost o critério de classificação mais significativo. Um sinal de subtração a frente (-) indica uma classificação em ordem decrescente daquele elemento. Você pode ter vários parâmetros de valor para indicar os tipos na classificação. É possível usar as palavras-chave RECORD para indicar uma classificação na ordem crescente em todos os campos, e/ou usar a palavra-chave EXCEPT para listar os campos que não serão classificados no recordset.
<i>index</i>	O nome de atributo de uma definição INDEX. Isso é equivalente a adicionar a opção SORTED para a definição INDEX.
Return:	SORTED é uma diretiva de compilador que não retorna nada.

A função **SORTED** indica para o compilador ECL que o *recordset* já está classificado segundo os *valores* especificados. É possível definir qualquer quantidade de parâmetros de *valor*, sendo o leftmost o critério de classificação mais importante. Um sinal negativo (-) à esquerda de qualquer parâmetro de *valor* indica uma ordem decrescente para o parâmetro. SORTED normalmente se refere a um DATASET para indicar a ordem na qual os dados já estão classificados.

Exemplo:

```
InputRec := RECORD
INTEGER4 Attr1;
STRING20 Attr2;
INTEGER8 Cid;
END;
MyFile := DATASET('filename',InputRec,FLAT)
MySortedFile := SORTED(MyFile,MyFile.Cid)
// Input file already sorted by Cid
```

Ver também: SORT, DATASET, RANK, RANKED, INDEX

SQRT

SQRT(*n*)

n	O número real que será avaliado.
Return:	SQRT retorna um valor real único.

A função **SQRT** retorna a raiz quadrada do parâmetro.

Exemplo:

```
MyRoot := SQRT(16.0);
```

Ver também: POWER, EXP, LN, LOG

STEPPED

STEPPED(*index*, *fields* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>index</i>	O INDEX para classificar. Pode ser filtrado ou o resultado de um PROJECT em um INDEX.
<i>fields</i>	Uma lista de campos delimitada por vírgulas pela qual o resultado é classificado, normalmente com elementos à esquerda na chave.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.

A função **STEPPED** classifica o *índice* pelos *campos* especificados. Essa função é usada em casos nos quais a função SORTED (índice) não será suficiente.

Há algumas restrições para o seu uso:

Os campos de chave anteriores aos *campos* ordenados devem estar razoavelmente bem filtrados; caso contrário, a classificação pode consumir muita memória.

O Roxie só oferece suporte para classificação por componentes à esquerda ou índices que são lidos localmente (índices de parte única ou superchaves que contenham índices de parte única) ou ainda índices NOROOT lidos em ALLNODES.

O Thor não oferece suporte para STEPPED.

Exemplo:

```
DataFile := '~RTTEST::TestStepped';
KeyFile := '~RTTEST::TestSteppedKey';
Rec := RECORD
  STRING2 state;
  STRING20 city;
  STRING25 lname;
  STRING15 fname;
END;
ds := DATASET(DataFile,
  {Rec, UNSIGNED8 RecPos {virtual(fileposition)}} ,
  THOR);
IDX := INDEX(ds, {state, city, lname, fname, RecPos}, KeyFile);

OUTPUT(IDX(state IN ['FL', 'PA']));
/* where this OUTPUT produces this result:
FL BOCA RATON WIK PICHA
FL DELAND WIKER OKE
FL GAINESVILLE WIK MACHOUSTON
```

```
PA NEW STANTON WIKER DESSIE */

OUTPUT(STEPPED(IDX(state IN ['FL','PA']),fname));
/* this STEPPED OUTPUT produces this result:
PA NEW STANTON WIKER DESSIE
FL GAINESVILLE WIK MACHOUSTON
FL DELAND WIKER OKE
FL BOCA RATON WIK PICHA */
```

Ver também: INDEX, SORTED, ALLNODES

STORED

STORED(*interface*)

interface O nome de um atributo de estrutura INTERFACE.

A função **STORED** é uma forma de atalho da definição de atributos a ser usada em uma interface SOAP. É equivalente a definição de uma estrutura **MODULE** que herda todos os atributos da *interface* e adiciona o serviço de fluxo de trabalho **STORED** para cada atributo, usando o nome do atributo como nome **STORED**.

Exemplo:

```
Iname := INTERFACE
EXPORT STRING20 Name;
EXPORT BOOLEAN KeepName := TRUE;
END;

StoredName := STORED(Iname);
// is equivalent to:
// StoredName := MODULE(Iname)
// EXPORT STRING20 Name := '' : STORED('name');
// EXPORT BOOLEAN KeepName := TRUE : STORED('keepname');
// END;
```

Ver também: **STORED Workflow Service**, Estrutura **INTERFACE**, Estrutura **MODULE**

SUM

SUM(*recordset*, *value*, [, *expression*] [, **KEYED**])

SUM(*valuelist* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Também pode ser a palavra-chave GROUP para indicar a soma dos valores do campo em um grupo, quando usada em uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>value</i>	A expressão a ser somada.
<i>expression</i>	Opcional. Uma expressão lógica indicando quais registros devem ser incluídos na soma. Válido apenas quando o parâmetro <i>recordset</i> for a palavra-chave GROUP para indicar a soma dos elementos em um grupo.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
<i>valuelist</i>	Uma lista delimitada por vírgula das expressões cuja soma será encontrada. Também pode ser um SET de valores.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	SUM retorna um único valor.

A função **SUM** retorna a soma aditiva do *valor* em cada registro do *recordset* ou *valuelist*.

Exemplo:

```
MySum := SUM(Person,Person.Salary); // total all salaries

SumVal2 := SUM(4,8,16,2,1); //returns 31
SetVals := [4,8,16,2,1];
SumVal3 := SUM(SetVals); //returns 31
```

Ver também: COUNT, AVE, MIN, MAX

TABLE

TABLE(recordset, format [, expression [,FEW | MANY] [, UNSORTED]] [, LOCAL] [, KEYED] [, MERGE] [, SKEW(limit[, target])] [, THRESHOLD(size)]] [, UNORDERED | ORDERED(bool)] [, STABLE | UNSTABLE] [, PARALLEL [(numthreads)]] [, ALGORITHM(name)])

<i>recordset</i>	O conjunto para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado.
<i>format</i>	Uma definição de estrutura RECORD que define o tipo, o nome e a fonte de dados para cada campo.
<i>expression</i>	Opcional. Especifica uma cláusula "agrupar por". É possível ter mais de uma expressão separada por vírgula para criar uma cláusula de "agrupar por" lógica e única. Se a expressão for um campo do recordset, então há um registro de grupo único na tabela resultante para cada valor distinto da expressão. Do contrário, a expressão é uma expressão de tipo LEFT/RIGHT no modo de DEDUP.
FEW	Opcional. Indica que a expressão resultará em menos de 10.000 grupos distintos. Isso permite otimização para gerar um resultado significativamente mais rápido.
MANY	Opcional. Indica que a expressão resultará em vários grupos distintos.
UNSORTED	Opcional. Especifica que você não se importa com a ordem dos grupos. Isso permite otimização para gerar um resultado significativamente mais rápido.
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
MERGE	Opcional. Especifica que os resultados são agregados em cada nó e depois os intermediários agregados são novamente agregados globalmente. Esse é um método seguro de agregação que se destaca especialmente bem se os dados adjacentes tiverem sido distorcidos. Se souber que o número de grupos será baixo, então FEW será ainda mais rápido, evitando a classificação local dos dados subjacentes.
SKEW	Indica que você sabe que os dados não serão espalhados uniformemente entre os nós (serão distorcidos e você opta por substituir o padrão especificando seu próprio valor limite para permitir que a tarefa continue, apesar da distorção).
<i>limit</i>	Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima de distorção a ser permitida antes que a tarefa falhe (a distorção padrão é 1,0 / <número de escravos no cluster>).
<i>target</i>	Opcional. Um valor entre zero (0) e um (1,0 = 100%) indicando a porcentagem máxima de distorção desejada a ser permitida (a distorção padrão é 1,0 / <número de escravos no cluster>).
THRESHOLD	Indica o tamanho mínimo de uma única parte antes que o limite SKEW seja aplicado.
<i>size</i>	Um valor inteiro indicando o número mínimo de bytes para uma parte única. O padrão é 1.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for "False" (Falso), especifica que a ordem do registro de resultado não é importante. Quando for "True" (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.

Referência a Linguagem ECL

Ações e Funções Built-in

PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads.
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	TABLE retorna uma nova tabela.

A função **TABLE** é similar a OUTPUT, mas, em vez de gravar registros em um arquivo, ela salva esses registros em uma nova tabela (um novo dataset no supercomputador) na memória. A nova tabela é temporária e existe apenas enquanto a consulta específica que a invocou está em execução.

A nova tabela herda a lógica implícita que o *recordset* possui (se aplicável), exceto caso a *expressão* opcional seja usada para realizar a agregação. Isso significa que o registro primário está disponível ao processar registros de tabela e que você também pode acessar o conjunto de registros secundários de cada registro de tabela. Há duas formas de usar TABLE: a forma de "fatia vertical" e a de "relatório de referência cruzada".

Para a forma de "fatia vertical", não há um parâmetro de *expressão* especificado. O número de registros no *recordset* de entrada é igual ao número de registros produzidos.

Para a forma de "relatório de referência cruzada", normalmente há um parâmetro de *expressão* e, o mais importante, a estrutura *RECORD do formato* de resultado contém, no mínimo, um campo que usa uma função agregada com a palavra-chave GROUP como seu primeiro parâmetro. O número de registros produzidos é igual ao número de valores distintos da *expressão*.

Exemplo:

```
// "vertical slice" form:
MyFormat := RECORD
  STRING25 Lname := Person.per_last_name;
  Person.per_first_name;
  STRING5 NewField := '';
END;
PersonTable := TABLE(Person, MyFormat);
// adding a new field is one use of this form of TABLE

// "CrossTab Report" form:
rec := RECORD
  Person.per_st;
  StCnt := COUNT(GROUP);
END
Mytable := TABLE(Person, rec, per_st, FEW);
// group persons by state in Mytable to produce a
  crosstab
```

Ver também: OUTPUT, GROUP, DATASET, Estrutura RECORD

TAN

TAN(*angle*)

<i>angle</i>	O valor radiano REAL para o qual será localizada a tangente.
Return:	TAN retorna um valor REAL único.

A função **TAN** retorna a tangente do *ângulo*.

Exemplo:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
Tangent45 := TAN(Angle45);  //get tangent of the 45 degree angle
```

Ver também: ACOS, COS, ASIN, SIN, ATAN, COSH, SINH, TANH

TANH

TANH(*angle*)

<i>angle</i>	O valor radiano REAL para o qual será localizada a tangente hiperbólica.
Return:	TANH retorna um valor REAL único.

A função **TANH** retorna a tangente hiperbólica do *ângulo*.

Exemplo:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
HyperbolicTangent45 := TANH(Angle45);
                        //get hyperbolic tangent of the angle
```

Ver também: ACOS, COS, ASIN, SIN, ATAN, COSH, SINH, TAN

THISNODE

THISNODE(*operation*)

<i>operation</i>	O nome de um atributo ou código em linha que resulta em um DATASET ou INDEX.
Return:	THISNODE retorna um conjunto de registros ou índice.

A função **THISNODE** especifica que a *operação* é realizada em cada nó de forma independente. Ela é geralmente usada em uma operação ALLNODES . **Essa função está disponível para uso apenas no Roxie.**

Exemplo:

```
ds := ALLNODES ( JOIN ( THISNODE ( GetData ( SomeData ) ) ,  
                        THISNODE ( GetIDX ( SomeIndex ) ) ,  
                        LEFT.ID = RIGHT.ID ) );
```

Ver também: ALLNODES, LOCAL, NOLOCAL

TOJSON

TOJSON(*record*)

<i>record</i>	A linha (registro) de dados a ser convertida para o formato JSON .
Return:	TOJSON retorna UTF8.

A função **TOJSON** retorna uma única string UTF8 com os dados no *registro* formatados novamente como JSON. Se a estrutura RECORD do *registro* tiver XPATHs definidos, eles seriam então usados; caso contrário, os nomes de campo em caixa baixa serão usados como nomes de tag JSON .

Exemplo:

```
namesRec1 := RECORD
  UNSIGNED2 EmployeeID{xpath('EmpID')};
  STRING10 Firstname{xpath('FName')};
  STRING10 Lastname{xpath('LName')};
END;
str1 := TOJSON(ROW({42,'Fred','Flintstone'},namesRec1));
OUTPUT(str1);
//returns this string:
//{"EmpID": 42, "FName": "Fred", "LName": "Flintstone"}

namesRec2 := RECORD
  UNSIGNED2 EmployeeID;
  STRING10 Firstname;
  STRING10 Lastname;
END;
str2 := TOJSON(ROW({42,'Fred','Flintstone'},namesRec2));
OUTPUT(str2);
//returns this string:
//{"employeeid": 42, "firstname": "Fred", "lastname": "Flintstone"}
```

Ver também: ROW, FROMJSON

TOPN

TOPN(*recordset*, *count*, *sorts* [, **BEST**(*bestvalues*)] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um record set derivado de algumas condições de filtro, ou qualquer expressão que resulte em um record set derivado.
<i>COUNT</i>	Uma expressão de valor inteiro que define o número de registros a serem retornados.
<i>sorts</i>	Uma lista delimitada por vírgula das expressões ou dos campos principais no recordset em que ocorrerá a classificação, sendo leftmost o critério de classificação mais significativo. Um sinal de subtração a frente (-) indica uma classificação em ordem decrescente daquele elemento. É possível usar as palavras-chave RECORD para indicar uma classificação na ordem crescente em todos os campos, e/ou usar a palavra-chave EXCEPT para listar os campos que não serão classificados no recordset.
BEST	Opcional. Permite terminar antecipadamente a operação se houver número de contagem de registros e se os valores contidos no último registro for correspondente aos <i>bestvalues</i> .
<i>bestvalues</i>	Uma lista delimitada por vírgula correspondente a lista de classificações dos valores máximos (ou mínimos se a classificação correspondente for decrescente).
LOCAL	Opcional. Especifica que a operação é realizada em cada nó de supercomputador de forma independente, sem exigir interação com todos os outros nós para obter dados; a operação mantém a distribuição de qualquer operação DISTRIBUTE anterior.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT.
Return:	TOPN retorna um recordset.

A função **TOPN** retorna o primeiro número de *contagem* de registros na ordem das *classificações* do *recordset*. Isto é quase equivalente ao **CHOOSEN**(**SORT**(*recordset*,*sorts*),*count*), mas com uma sintaxe mais simples. Também retorna os primeiros números de linhas em cada grupo de *recordsets* GROUPed e operações locais.

Exemplo:

```
y := TOPN(Person,1000,state,sex); //first 1000 recs in state, sex order
z := TOPN(Person,1000,sex,BEST('F')); //first 1000 females
```

Ver também: **CHOOSEN**, **SORT**, **GROUP**

TOUNICODE

TOUNICODE(*string*, *encoding*)

<i>string</i>	A string DATA para tradução.
<i>encoding</i>	A página do código de codificação (suportada pelo ICU da IBM) a ser usada para tradução.
Return:	TOUNICODE retornos um único valor UNICODE.

A função **TOUNICODE** retorna a *string* traduzida do valor de DADOS DATA para a codificação unicode *especificada*.

Exemplo:

```
DATA5 x := FROMUNICODE(u'ABCDE', 'UTF-8');  
      //results in 4142434445  
y := TOUNICODE(x, 'US-ASCII');
```

Ver também: FROMUNICODE, UNICODEORDER

TOXML

TOXML(*record*)

<i>record</i>	A linha (registro) de dados a ser convertida para um formato XML.
Return:	TOXML retorna uma STRING UTF8.

A função **TOXML** retorna uma única string UTF-8 com os dados no *registro* reformatados como XML. Se a estrutura **RECORD** do *registro* tiver **XPATHs** definidos, eles serão usados, caso contrário, os nomes de campo com menos letras maiúsculas serão usados como nomes de tag XML.

Exemplo:

```
namesRec1 := RECORD
  UNSIGNED2 EmployeeID{xpath('EmpID')};
  STRING10  Firstname{xpath('FName')};
  STRING10  Lastname{xpath('LName')};
END;
rec1 := TOXML(ROW({42, 'Fred', 'Flintstone'}, namesRec1));
OUTPUT(rec1);

//returns this string:
// '<EmpID>42</EmpID><FName>Fred</FName><LName>Flintstone</LName>'

namesRec2 := RECORD
  UNSIGNED2 EmployeeID;
  STRING10  Firstname;
  STRING10  Lastname;
END;
rec2 := TOXML(ROW({42, 'Fred', 'Flintstone'}, namesRec2));
OUTPUT(rec2);

//returns this string:
// '<employeeid>42</employeeid><firstname>Fred</firstname><lastname>Flintstone</lastname>'
```

Ver também: **ROW**, **FROMXML**

TRACE

[*attrname* :=] **TRACE**(*baserecset*, [*options*]);

<i>attrname</i>	Opcional. O nome para a expressão.
<i>baserecset</i>	O conjunto de registros de dados para o qual TRACE será definido.
<i>options</i>	Opcional. Uma ou mais das opções listadas abaixo.

A expressão **TRACE** define o rastreamento para os arquivos de log (logs escravos do Thor, logs do hThor logs, ou logs do Roxie).

Você pode adicionar TRACE ao seu código em pontos críticos sem que o desempenho sofra nenhum impacto. Posteriormente, se precisar investigar o comportamento, é possível habilitá-lo sem modificar o código ao definir uma opção ou o BOOLEAN armazenado.

O rastreamento é codificado aos arquivos de log na seguinte forma:

```
TRACE: <name><fieldname>value</fieldname>...</name>
```

O rastreamento não é gerado por padrão, mesmo que as declarações TRACE estejam presentes; o rastreamento é gerado apenas quando o valor de depuração da tarefa traceEnabled está configurado ou se as configurações padrão da plataforma forem alteradas para “sempre gerar rastreamento”. No Roxie, você também pode solicitar o rastreamento em uma consulta implementada especificando traceEnabled=1 na consulta XML.

Consequentemente, é possível deixar as declarações TRACE no ECL sem nenhuma sobrecarga detectável até que o rastreamento seja ativado. Para ativar o rastreamento:

```
#OPTION ('traceEnabled', 1) // trace statements enabled
```

Também é possível substituir o valor padrão de KEEP em nível global, por workunit ou por consulta.

```
#OPTION ('traceLimit', 100) // overrides the default KEEP value (10)
```

É possível usar um BOOLEAN armazenado como a expressão de filtro de uma atividade de rastreamento para permitir ativar e desativar atividades individuais de rastreamento.

Opções TRACE

As seguintes opções estão disponíveis para TRACE:

[*filterExpression*,] [**KEEP**(*n*),] [**SKIP**(*n*),] [**SAMPLE**(*n*),][**NAMED**(*string*)]

<i>filterExpression</i>	Opcional. Uma expressão válida que age como um filtro. Apenas as linhas correspondentes à condição do filtro são incluídas no rastreamento.
KEEP (<i>n</i>)	Opcional. Especifica o número de linhas para rastrear.
SKIP (<i>n</i>)	Opcional. Especifica o número de linhas que devem ser puladas antes do início do rastreamento.
SAMPLE (<i>n</i>)	Opcional. Especifica que apenas as linhas <i>nth</i> são rastreadas.
NAMED (<i>string</i>)	Opcional. Especifica o nome das linhas em rastreamento.

Exemplo:

```
#OPTION ('traceEnabled', TRUE); //TRACE writes to log only if TRUE
FilterValue := 4;
myRec := { STRING Name, REAL x, REAL y };
ds := DATASET([ {'Jim' , 1, 1.00039},
                {'Jane', 2, 2.07702},
                {'Emil', 3, 2.86158},
                {'John', 4, 3.87114},
                {'Jean', 5, 5.12417},
                {'Gene', 6, 6.20283} ], myRec);
myds := TRACE(ds,x>filterValue,NAMED('person')); //trace only if x > filterValue
myds;
```

TRANSFER

TRANSFER(*value,type*)

<i>value</i>	Uma expressão que contém o bitmap para retornar.
<i>type</i>	O tipo de valor para retornar.
Return:	TRANSFER retorna um valor único.

A função **TRANSFER** retorna o *valor* no *tipo* solicitado. Essa não é uma conversão de tipo porque o padrão de bits permanece inalterado.

Exemplo:

```
INTEGER1 MyInt := 65; //MyInt is an integer whose value is 65
STRING1 MyVal := TRANSFER(MyInt,STRING1); //MyVal is "A" (ASCII 65)
INTEGER1 MyVal2 := (INTEGER)MyVal; //MyVal2 is 0 (zero) because
    "A" is not a numeric character
```

Ver também: Type Casting

TRIM

TRIM(*string_value* [, *flag*])

<i>string_value</i>	A string da qual os espaços serão removidos.
<i>flag</i>	Opcional. Especifica quais espaços serão removidos. Os valores válidos do indicador são: RIGHT (remove espaços à direita – este é o padrão), LEFT (remove espaços à esquerda), LEFT, RIGHT (remove espaços à direita e à esquerda), e ALL (remove todos os espaços, até mesmo aqueles dentro de <i>string_value</i>).
Return:	TRIM retorna um único valor.

A função **TRIM** retorna o *string_value* com todos os espaços à direita e/ou à esquerda removidos.

Exemplo:

```
STRING20 SomeStringValue := 'ABC';  
//contains 17 trailing spaces  
  
VARSTRING MyVal := TRIM(SomeStringValue);  
// MyVal is "ABC" with no trailing spaces  
  
STRING20 SomeStringValue := ' ABC DEF';  
//contains 2 leading and 11 trailing spaces  
  
VARSTRING MyVal := TRIM(SomeStringValue,LEFT,RIGHT);  
// MyVal is "ABC DEF" with no trailing spaces
```

Ver também: STRING, VARSTRING

TRUNCATE

TRUNCATE(*real_value*)

<i>real_value</i>	O valor de ponto flutuante a ser truncado.
Return:	TRUNCATE retorna um valor inteiro único.

A **função** TRUNCATE retorna a parte inteira de *real_value*.

Exemplo:

```
SomeRealValue := 3.75;  
INTEGER4 MyVal := TRUNCATE(SomeRealValue); // MyVal is 3
```

Ver também: ROUND, ROUNDUP

UNGROUP

UNGROUP(*recordset* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	O conjunto de registros previamente AGRUPADOS GROUPed.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	UNGROUP retorna um conjunto de registros.

A **função** UNGROUP remove um agrupamento prévio. Isto equivale ao uso da função **GROUP** sem um segundo parâmetro.

Exemplo:

```
MyRec := RECORD
  STRING20 Last;
  STRING20 First;
END;

SortedSet := SORT(Person, Person.last_name); //sort by last
name
GroupedSet := GROUP(SortedSet, last_name); //then group
them

SecondSort := SORT(GroupedSet, Person.first_name);
//sorts by first name within each last name group
// this is a "sort within group"

UnGroupedSet := UNGROUP(GroupedSet); //ungroup the
dataset
```

Ver também: **GROUP**

UNICODEORDER

UNICODEORDER(*left*, *right* [, *locale*])

<i>left</i>	A expressão Unicode left a ser avaliada.
<i>right</i>	A expressão Unicode right a ser avaliada.
<i>locale</i>	Opcional. Uma constante de string que contém um código de localidade válido, como especificado nas normas ISO 639 e 3166.
Return:	UNICODEORDER retorna um único valor.

A **função** UNICODEORDER retorna -1, 0, ou 1, dependendo da avaliação das expressões *left* e *right*. Isto equivale ao operador de comparação de equivalência \leq , porém adotando o unicode *locale* como base de determinação. Se *left* < *right*, será retornado o valor -1; se *left* = *right*, será retornado valor 0; se *left* > *right*, será retornado o valor 1.

Exemplo:

```
UNICODE1 x := u'a';
UNICODE1 y := u'b';
UNICODE1 z := u'a';

a := UNICODEORDER(x , y, 'es'); // returns -1
b := UNICODEORDER(x , z, 'es'); // returns 0
c := UNICODEORDER(y , z, 'es'); // returns 1
```

Ver também: FROMUNICODE, TOUNICODE

UNORDERED

UNORDERED(*dataset*)

<i>dataset</i>	O nome do DATASET desordenado.
----------------	--------------------------------

A função **UNICODEORDER** é utilizada para indicar que a ordem do *Dataset* não é importa. Isso permitirá que o gerador de código aplique otimizações adicionais em futuras versões.

Exemplo:

```
Def1 := UNORDERED(MyDataset);  
//the order of MyDataset is not significant,  
//so the code generator can perform optimizations based on that
```

VARIANCE

VARIANCE(*recset*, *valuex* [, *expression*] [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recset</i>	O conjunto de registros para processamento. Pode ser o nome de um dataset ou de um recordset derivado de algumas condições de filtro, ou qualquer expressão que resulte em um recordset derivado. Isto também pode ser a palavra-chave GROUP para indicar a operação sobre os elementos em cada grupo quando usada em uma estrutura RECORD para gerar estatísticas de tabela de referência cruzada.
<i>valuex</i>	Um campo ou expressão numérica.
<i>expressão</i>	Opcional. Uma expressão lógica indicando quais registros devem ser incluídos no cálculo. Válido apenas quando o parâmetro <i>recset</i> for a palavra-chave GROUP .
KEYED	Opcional. Especifica que a atividade faz parte de uma operação de leitura de índice, a qual permite que o otimizador gere o código ideal para a operação.
UNORDERED	Opcional. Especifica que a ordem do registro de resultado não é importante.
ORDERED	Especifica a importância da ordem do registro de resultado.
<i>bool</i>	Quando for “False” (Falso), especifica que a ordem do registro de resultado não é importante. Quando for “True” (Verdadeiro), especifica a ordem padrão do registro de resultado.
STABLE	Opcional. Especifica que a ordem do registro de entrada é importante.
UNSTABLE	Opcional. Especifica que a ordem do registro de entrada não é importante.
PARALLEL	Opcional. Tenta avaliar essa atividade em paralelo.
<i>numthreads</i>	Opcional. Tenta avaliar essa atividade usando os <i>numthreads</i> threads
ALGORITHM	Opcional. Substitui o algoritmo usado para essa atividade.
<i>name</i>	O algoritmo a ser usado para essa atividade. Precisa fazer parte da lista de algoritmos compatíveis com as opções STABLE e UNSTABLE da função SORT .
Return:	VARIANCE retorna um valor real único.

A função **VARIANCE** retorna a variância (população) de *valuex*.

Exemplo:

```
pointRec := { REAL x, REAL y };

analyse( ds ) := MACRO

#uniqueName(stats)
%stats% := TABLE(ds, { c := COUNT(GROUP),
sx := SUM(GROUP, x),
sy := SUM(GROUP, y),
sxx := SUM(GROUP, x * x),
sxy := SUM(GROUP, x * y),
syy := SUM(GROUP, y * y),
varx := VARIANCE(GROUP, x);
vary := VARIANCE(GROUP, y);
varxy := COVARIANCE(GROUP, x, y);
rc := CORRELATION(GROUP, x, y) });
OUTPUT(%stats%);

// Following should be zero
OUTPUT(%stats%, { varx - (sxx-sx*sx/c)/c,
```

```
vary - (syy-sy*sy/c)/c,  
varxy - (sxy-sx*sy/c)/c,  
rc - (varxy/SQRT(varx*vary)) });  
  
OUTPUT(%stats%, { 'bestFit: y=' +  
(STRING)((sy-sx*varxy/varx)/c) +  
' + ' +  
(STRING)(varxy/varx)+'x' });  
ENDMACRO;  
ds1 := DATASET([ {1,1}, {2,2}, {3,3}, {4,4}, {5,5}, {6,6} ],  
    pointRec);  
  
ds2 := DATASET([ {1.93896e+009, 2.04482e+009},  
    {1.77971e+009, 8.54858e+008},  
    {2.96181e+009, 1.24848e+009},  
    {2.7744e+009, 1.26357e+009},  
    {1.14416e+009, 4.3429e+008},  
    {3.38728e+009, 1.30238e+009},  
    {3.19538e+009, 1.71177e+009} ], pointRec);  
  
ds3 := DATASET([ {1, 1.00039},  
    {2, 2.07702},  
    {3, 2.86158},  
    {4, 3.87114},  
    {5, 5.12417},  
    {6, 6.20283} ], pointRec);  
  
analyse(ds1);  
analyse(ds2);  
analyse(ds3);
```

Ver também: CORRELATION, COVARIANCE

WAIT

WAIT(*event*)

<i>event</i>	Uma constante de string com o nome do evento a ser aguardado.
--------------	---

A ação **WAIT** é similar ao serviço de fluxo de trabalho **WHEN**, mas pode ser usada no código condicional.

Exemplo:

```
//You can either do this:
action1;
action2 : WHEN('expectedEvent');

//can also be written as:
SEQUENTIAL(action1, WAIT('expectedEvent'), action2);
```

Ver também: **EVENT**, **NOTIFY**, **WHEN**, **CRON**

WHEN

WHEN(*trigger*, *action* [, **BEFORE** | **SUCCESS** | **FAILURE**])

<i>trigger</i>	Um dataset ou ação que inicializa a <i>ação</i> .
<i>action</i>	A ação a ser executada.
BEFORE	Opcional. Especifica uma <i>ação</i> que deve ser executada antes da leitura da entrada.
SUCCESS	Opcional. Especifica uma <i>ação</i> específica uma ação que só deve ser executada mediante ao SUCCESS do <i>trigger</i> (p.ex., quando os LIMITES não são excedidos).
FAILURE	Opcional. Especifica uma <i>ação</i> que só deve ser executada mediante a FALHA FAILURE do <i>acionador</i> (p.ex., quando o LIMITE LIMIT é excedido).

A função **WHEN** associa uma *ação* a um *acionador* (dataset ou ação) para que quando o *acionador* for executado, a *ação* também seja executada. Isso permite o agendamento de tarefas com base em acionadores.

Exemplo:

```
//a FUNCTION with side-effect Action
namesTable := FUNCTION
  namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
  END;
  o := OUTPUT('namesTable used by user <x>');
  ds := DATASET([{'x','y',22}],namesRecord);
  RETURN WHEN(ds,O);
END;

z := namesTable : PERSIST('z');
//the PERSIST causes the side-effect action to execute only when the PERSIST is re-built
OUTPUT(z);
```

Ver também: Estrutura FUNCTION, WHEN, WAIT

WHICH

WHICH(*condition*,...,*condition*)

<i>condition</i>	Uma expressão condicional para avaliar.
Return:	WHICH retorna um único valor.

A função **WHICH** avalia quais das listas de *condições* retornaram como “true” (verdadeiras) e retorna sua posição ordinal na lista de *condições*. Se nenhuma retornar como verdadeira, é indicado zero (0). Trata-se do oposto da função REJECTED.

Exemplo:

```
Accept := WHICH(Person.per_first_name = 'Fred',  
Person.per_first_name = 'Sue');  
//Accept is 0 for everyone but those named Fred or Sue
```

Ver também: REJECTED, MAP, CHOOSE, IF, CASE

WORKUNIT

WORKUNIT

WORKUNIT(*named* [, *type*])

<i>named</i>	Uma constante de string que contém o valor scalar da opção NAMED a ser retornado.
<i>type</i>	Opcional. O tipo de valor do resultado do valor scalar nomeado a ser retornado.
Return:	Workunit retorna um único valor.

A função **WORKUNIT** retorna os valores armazenados na workunit. Dado nenhum parâmetro, ele retorna o identificador de workunit exclusivo (WUID) para a workunit que está sendo executada atualmente; caso contrário, ela retorna o resultado da opção NAMED do OUTPUT ou da ação DISTRIBUTION .

Exemplo:

```
wuid := WORKUNIT; //get WUID

namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age;
END;

namesTable := DATASET([{'Halligan','Kevin',31},
                        {'Halligan','Liz',30},
                        {'Salter','Abi',10},
                        {'X','Z',42}], namesRecord);

DISTRIBUTION(namesTable, surname, forename,NAMED('Stats'));
WORKUNIT('Stats',STRING);
```

Ver também: #WORKUNIT, OUTPUT, DISTRIBUTION

XMLDECODE

XMLDECODE(*unicode*)

<i>unicode</i>	O texto unicode a ser decodificado.
Return:	XMLDECODE retorna um único valor.

A função **XMLDECODE** decodifica caracteres especiais em uma string XML (por exemplo, < é convertido para <) para permitir o uso da opção CSV no OUTPUT, a fim de gerar arquivos XML mais complexos em comparação ao uso da opção XML.

Exemplo:

```
d := XMLENCODE('<xml version 1><tag>data</tag>');  
e := XMLDECODE(d);
```

Ver também: XMLENCODE

XMLENCODE

XMLENCODE(*xml* [, **ALL**])

<i>xml</i>	O XML para codificar.
ALL	Opcional. Especifica a inclusão de novos caracteres de linha na codificação para que o texto possa ser usado em definições de atributo.
Return:	XMLENCODE retorna um valor único.

A função **XMLENCODE** codifica caracteres especiais em uma string XML (por exemplo, < é convertido para <) para permitir o uso da opção CSV no OUTPUT, a fim de gerar arquivos XML mais complexos em comparação ao uso da opção XML.

Exemplo:

```
d := XMLENCODE('<xml version 1><tag>data</tag>');  
e := XMLDECODE(d);
```

Ver também: XMLDECODE

Serviços de Fluxo de Trabalho

Visão Geral do Fluxo de Trabalho

O controle do fluxo de trabalho no ECL é geralmente manuseado automaticamente pelo sistema. Ele distingue quais processos podem ser realizados em paralelo, quando a sincronização é exigida, e quando os processos devem ser realizados em série. Estes serviços de fluxo de trabalho permitem que o programador especifique as exceções ao fluxo normal de execução para oferecer controle adicional (tal como a cláusula FAILURE).

As operações de fluxo de trabalho são avaliadas de forma implícita em um escopo global separado do código ao qual estão anexadas. Consequentemente, qualquer valor do código ao qual estão anexadas (como por exemplo os contadores loop) estará indisponível para o serviço de fluxo de trabalho.

Também deve ser observado que quando uma operação de fluxo de trabalho está presente dentro de múltiplas declarações SEQUENTIAL, apenas a primeira instância será avaliada.

Exemplo:

```
Chesney := OUTPUT('"I am the one and only!" said Chesney')
         : SUCCESS(OUTPUT('"Oh yeah, prove it?"'));

SEQUENTIAL(
    OUTPUT('"I am Spartacus" said one from the mob'),
    Chesney
);

SEQUENTIAL(
    OUTPUT('"No, I am Spartacus" confessed another'),
    Chesney,
    OUTPUT('"Ok, so you are!"')
);
```

yields:

```
"I am Spartacus" said one from the mob
"I am the one and only!" said Chesney
"Oh yeah, prove it?"
"No, I am Spartacus" confessed another
"Ok, so you are!"
```

Ver também: SEQUENTIAL

CHECKPOINT

attribute := *expression* : **CHECKPOINT**(*name*) ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>name</i>	Uma constante de string que especifica o nome de armazenamento do valor.

O serviço **CHECKPOINT** armazena o resultado de *expression* na unidade de trabalho para que permaneça disponível caso a tarefa falhe antes da conclusão, e é excluído automaticamente quando a tarefa é concluída de forma bem-sucedida. Isso é especialmente útil para *atributos* baseados em sequências de manipulação de dados dispendiosas e grandes. Esse serviço causa implicitamente a avaliação de *attribute* no escopo global em vez de qualquer escopo que o abrange.

No entanto, **CHECKPOINT** somente é útil quando a workunit que não concluiu corretamente é iniciada novamente por meio do ECL Watch. Se uma nova workunit for instanciada, **CHECKPOINT** não terá efeito nenhum. No geral, o serviço **PERSIST** é mais útil.

Exemplo:

```
CountPeople := COUNT(Person) : CHECKPOINT('PeopleCount');  
    //Makes CountPeople available for reuse if  
    // the job does not complete
```

Ver também: **PERSIST**

DEPRECATED

attribute := *expression* : **DEPRECATED** [(*message*)] ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>message</i>	Opcional. O texto a ser anexado ao aviso se o atributo for usado.

O serviço **DEPRECATED** exibirá um aviso quando *attribute* for usado em código que instancia uma tarefa ou durante uma verificação de sintaxe. O uso pretendido se limita a definições de atributos que foram suplantados.

Quando usado em um atributo de estrutura (RECORD, TRANSFORM, FUNCTION, etc.), deve ser posicionado entre a palavra-chave END e o seu caractere de ponto e vírgula.

Exemplo:

```
OldSort := SORT(Person, Person.per_first_name) : DEPRECATED('Use NewSort instead.');
```

```
NewSort := SORT(Person, -Person.per_first_name);
```



```
OUTPUT(OldSort);
```

```
//produces this warning:
```

```
// Attribute OldSort is marked as deprecated. Use NewSort instead.
```



```
//*****
```

```
ds := DATASET(['A', 'B', 'C'], {STRING1 letter});
```



```
R1 := RECORD
```

```
    STRING1 letter;
```

```
END : DEPRECATED('Use R2 now.');
```



```
R2 := RECORD
```

```
    STRING1 letter;
```

```
    INTEGER number;
```

```
END;
```



```
R1 Xform1(ds L) := TRANSFORM
```

```
    SELF.letter := Std.Str.ToLowerCase(L.letter);
```

```
END : DEPRECATED('Use Xform2 now.');
```



```
R2 Xform2(ds L, integer C) := TRANSFORM
```

```
    SELF.letter := Std.Str.ToLowerCase(L.letter);
```

```
    SELF.number := C;
```

```
END;
```



```
OUTPUT(PROJECT(ds, Xform1(LEFT))); //produces these warnings:
```

```
// Attribute r1 is marked as deprecated. Use R2 now.
```

```
// Attribute Xform1 is marked as deprecated. Use Xform2 now.
```

FAILURE

attribute := *expression* : **FAILURE**(*handler*) ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>handler</i>	A ação a ser executada se a expressão falhar.

O serviço **FAILURE** executa a *manipulação* do atributo quanto a *expressão* falha. Teoricamente, FAILURE executa em paralelo com o retorno com falha do resultado. Esse serviço implicitamente faz com que o *atributo* seja avaliado em um escopo global em vez de escopo de função. Disponível apenas se os serviços de fluxo de trabalho estiverem ativados (consulte #OPTION(workflow)).

Exemplo:

```
sPeople := SORT(Person, Person.per_first_name);
nUniques := COUNT(DEDUP(sPeople, Person.per_first_name AND
                        Person.address))
          : FAILURE(Email.simpleSend(SystemsPersonel,
                                     SystemsPersonel.email, 'ouch.htm'));
```

Ver também: SUCCESS, RECOVERY

GLOBAL - Serviço

attribute := *expression* : **GLOBAL** [(*cluster* [, **FEW**])];

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>cluster</i>	Opcional. Uma constante de string que especifica o nome do cluster de supercomputadores onde será criado o atributo. Isso possibilita usar o atributo em um cluster menor, quando ele deve ser criado em um cluster maior para permitir um uso mais eficiente dos recursos. Se omitida, o atributo será criado no cluster em execução no momento.
FEW	Opcional. Quando a expressão é um dataset ou conjunto de recursos, FEW especifica que o dataset resultante é armazenado totalmente dentro da workunit. Se não especificada, o dataset será armazenado como um arquivo THOR e a workunit conterá apenas o nome arquivo.

O serviço **GLOBAL** faz com que *atributo* seja avaliado no escopo global, em vez de no escopo que o abrange, de forma semelhante à função GLOBAL(), ou seja, não dentro de um filtro/transformação, etc. Ele poderá ser avaliado várias vezes na mesma tarefa, se usado de vários itens do fluxo do trabalho, mas compartilhará o código com o contexto onde for usado.

GLOBAL opera de forma diferente de INDEPENDENT, pois INDEPENDENT é executado apenas uma vez e GLOBAL é executado uma vez em cada item de fluxo de trabalho que o usa.

Exemplo:

```
I := RANDOM() : INDEPENDENT; //calculated once, period
G := RANDOM() : GLOBAL;      //calculated once in each graph

ds :=
  DATASET([ {1,0,0,0}, {2,0,0,0} ],
    {UNSIGNED1 rec, UNSIGNED Ival, UNSIGNED Gval , UNSIGNED Aval });

RECORDOF(ds) XF(ds L) := TRANSFORM
  SELF.Ival := I;
  SELF.Gval := G;
  SELF.Aval := RANDOM(); //calculated each time used
  SELF := L;
END;

P1 := PROJECT(ds, XF(left)) : PERSIST('~RTTEST::PERSIST::IndependentVsGlobal1');
P2 := PROJECT(ds, XF(left)) : PERSIST('~RTTEST::PERSIST::IndependentVsGlobal2');

OUTPUT(P1);
OUTPUT(P2); //this gets the same Ival values as P1, but the Gval value is different than P1
```

Ver também: Função GLOBAL, INDEPENDENT

INDEPENDENT

attribute := *expression* : **INDEPENDENT** [(*cluster*)] ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>cluster</i>	Opcional. Uma constante de string que especifica o nome do cluster Thor em que o atributo será executado. Se omitida, o <i>atributo</i> é executado no cluster atual em execução.

O serviço **INDEPENDENT** faz com que o *atributo* seja avaliado em um escopo global e força a avaliação do *atributo* em um item de fluxo de trabalho individual. O novo item de fluxo de trabalho é avaliado antes do primeiro item de fluxo de trabalho que usa esse *atributo*. É executado de forma independente dos outros itens do fluxo de trabalho, e sua execução é realizada apenas uma vez (incluindo SEQUENTIAL onde deve ser executado na primeira vez que for usado). Ele não compartilhará nenhum código com nenhum outro item do fluxo de trabalho.

Uma forma de uso seria para fornecer um mecanismo de interface do código compartilhado entre os diferentes argumentos de uma ação SEQUENTIAL – eles são normalmente avaliados de forma totalmente independente.

Exemplo:

```
IMPORT STD;
File1 := 'names1.txt';
File2 := 'names2.txt';

SrcIP := '10.239.219.2';
SrcPath := '/var/lib/HPCCSystems/mydropzone/';
DestPath := '~THOR::IN::';
ESPportIP := 'http://192.168.56.120:8010/FileSpray';

DeleteOldFiles :=
  PARALLEL(STD.File.DeleteLogicalFile(DestPath+File1),
    STD.File.DeleteLogicalFile(DestPath+File2))
    : INDEPENDENT;

SprayNewFiles :=
  PARALLEL(STD.File.SprayFixed(SrcIP,SrcPath+File1,11,
    'mythor',DestPath+File1,
    -1,ESPportIP),
    STD.File.SprayFixed(SrcIP,SrcPath+File2,11,
    'mythor',DestPath+File2,
    -1,ESPportIP))
    : INDEPENDENT;

SEQUENTIAL(DeleteOldFiles,SprayNewFiles);
```

Ver também: GLOBAL

ONWARNING

attribute := *expression* : **ONWARNING**(*code*, *action*) ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>code</i>	O número exibido na coluna "Code" da caixa de ferramentas de erros de sintaxe do ECL IDE.
<i>action</i>	Uma destas ações: ignorar, erro ou aviso.

O serviço **ONWARNING** permite especificar como lidar com avisos específicos em relação a um determinado atributo. Você pode tratar a declaração como aviso, promovê-la a um erro ou ignorá-la. Avisos úteis podem ficar perdidos em um grande número de outros menos úteis. Esse recurso permite limpar a "confusão".

Este serviço substitui o manuseio de qualquer aviso global especificado por #ONWARNING.

Exemplo:

```
rec := { STRING x } : ONWARNING(1041, ignore);  
    //ignore "Record doesn't have an explicit maximum record size" warning
```

Ver também: #ONWARNING

PERSIST

attribute := *expression* : **PERSIST**(*filename* [, *cluster*] [, **CLUSTER**(*target*)] [, **EXPIRE**(*days*)] [, **REFRESH**(*flag*)] [, **SINGLE** | **MULTIPLE**[(*count*)]]) ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo. Isso normalmente define um recordset (mas pode ser qualquer expressão).
<i>filename</i>	Uma constante de string que especifica o nome de armazenamento do resultado de expressão. Consulte Escopo e nomes de arquivo lógicos .
<i>cluster</i>	Opcional. Uma constante de string que especifica o nome do cluster Thor no qual o <i>atributo</i> é recompilado se/quando necessário. Isso possibilita usar atributos persistentes em clusters menores, mas compilá-los em maiores, tornando a utilização de recursos mais eficiente. Se omitido, o <i>atributo</i> é recompilado no cluster em execução atualmente.
CLUSTER	Opcional. Especifica a gravação do <i>nome de arquivo</i> para a lista especificada de clusters de <i>destino</i> . Se omitido, o <i>nome de arquivo</i> é gravado no cluster no qual a função PERSIST é executada (como especificado pelo parâmetro <i>cluster</i>). O número de partes de arquivos físicos gravadas em disco é sempre determinado pelo número de nós no <i>cluster</i> no qual a função PERSIST é executada, independentemente do número de nós no(s) <i>destino(s)</i> .
<i>target</i>	Uma lista de constantes de string delimitada por vírgulas que contém os <i>nomes</i> dos clusters no qual o arquivo será gravado. Os nomes devem estar listados como aparecem na página de Atividade do ECL Watch, ou como são retornados pela função Std.System.Thorlib.Group(); opcionalmente, podem apresentar colchetes contendo uma lista delimitada por vírgula dos números dos nós (baseado em 1) e/ou dos intervalos (especificados com um traço, como p.ex., n-m) para indicar o conjunto específico de nós para gravar.
EXPIRE	Opcional. Especifica que o <i>nome de arquivo</i> é um arquivo temporário que pode ser removido automaticamente após um número especificado de dias.
<i>days</i>	Opcional. O número de dias em que o arquivo será automaticamente removido. Se omitido, por padrão, a configuração PersistExpiryDefault será usada no Sasha.
REFRESH	Opcional. Opção de controle quando o PERSIST é recompilado. Se omitido, o PERSIST recompila caso 1) o arquivo subjacente não exista, ou 2) os dados tenham sido alterados ou 3) o código tenha sido alterado.
<i>flag</i>	Um valor booleano indicando se PERSIST deve ser recompilado ou não. Quando definido para FALSE , o PERSIST recompila ONLY (APENAS) se o arquivo subjacente não existir. Se seu layout PERSIST tiver sido alterado e você especificar REFRESH(FALSE) , a incompatibilidade ocasionaria a falha do job.
SINGLE	Opcional. Especifica para manter um único PERSIST . O nome do arquivo de persistência é o mesmo do nome da persistência.
MULTIPLE	Opcional. Especifica para manter diferentes versões do PERSIST . O nome do arquivo de persistência gerado é uma combinação do nome fornecido com sufixo de valor de 32 bits derivado do ECL.
<i>count</i>	Opcional. O número de versões de um PERSIST deve ser mantido. Se omitido, o padrão do sistema será usado.

O serviço **PERSIST** armazena o resultado da *expressão* globalmente de forma a continuar permanentemente disponível para uso (incluindo o resultado de qualquer operação **DISTRIBUTE** ou **GROUP** na *expressão*). Isso é especialmente útil para *atributos* baseados em sequências de manipulação de dados dispendiosas e grandes. O *atributo* é recalculado apenas quando o código ECL ou os dados subjacentes que foram usados para criá-lo sofrem alterações; caso contrário,

os dados de *atributo* são simplesmente retornados do arquivo *nome* armazenado em disco quando referido. Implicitamente, esse serviço faz com que o *atributo* seja avaliado em um escopo global em vez de escopo de função.

PERSIST pode ser combinado com a cláusula WHEN para que, embora o *atributo* possa ser usado mais de uma vez, sua execução seja baseada na cláusula WHEN (ou o primeiro uso do *atributo*) e não baseada no número de vezes que o *atributo* é usado na computação. Isso oferece um tipo recurso denominado "computação antecipada".

Atributos persistentes ainda podem estar sujeitos aos requisitos de pedido de SEQUENTIAL. No entanto, como os PERSISTs são compartilhados entre as workunits, não há garantia de que o atributo será avaliado em qualquer ordem de avaliação fornecida.

Você pode usar #OPTION para sobrescrever as configurações padrão, assim como mostrado no exemplo.

Exemplo:

```
CountPeople := COUNT(Person) : PERSIST('PeopleCount');
//Makes CountPeople available for use in all subsequent work units

sPeople := SORT(Person, Person.per_first_name) :
    PERSIST('SortPerson'), WHEN(Daily);
//Makes sPeople available for use in all subsequent work units

s1 := SORT(Person, Person.per_first_name) :
    PERSIST('SortPerson1', 'OtherThor');
//run the code on the OtherThor cluster
s2 := SORT(Person, Person.per_first_name) :
    PERSIST('SortPerson2',
            'OtherThor',
            CLUSTER('AnotherThor'));
//run the code on the OtherThor cluster
// and write the file to the AnotherThor cluster
```

Ver também: STORED, WHEN, GLOBAL, CHECKPOINT

PRIORITY

action : **PRIORITY**(*value*) ;

<i>action</i>	Uma ação (geralmente OUTPUT) que irá gerar um resultado.
<i>value</i>	Um valor inteiro no intervalo de 0 a 100 indicando a importância relativa da ação.

O serviço **PRIORITY** determina a importância relativa de várias *ações* na workunit. Quanto mais alto for o *valor* de uma *ação* , maior será sua prioridade. A *ação* de maior prioridade será executada primeiro (se possível). **PRIORITY** não é permitida em definições de atributo; deve estar associada apenas com a ação. Disponível apenas se os serviços de fluxo de trabalho estiverem ativados (consulte #OPTION(workflow)).

Exemplo:

```
OUTPUT(Person(per_st='NY')) : PRIORITY(30);  
OUTPUT(Person(per_st='CA')) : PRIORITY(60);  
OUTPUT(Person(per_st='FL')) : PRIORITY(90);  
//The Florida
```

Ver também: OUTPUT, #OPTION

RECOVERY

attribute := *expression* : **RECOVERY**(*handler* [, *attempts*]) ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>handler</i>	A ação a ser executada se a expressão falhar.
<i>attempts</i>	Opcional. O número de tentativas antes da desistência.

O serviço **RECOVERY** executa o atributo *handler* quando a *expressão* falha e em seguida reexecuta o *atributo*. Se o *atributo* continuar falhando após determinado número de *tentativas*, qualquer cláusula **FAILURE** presente será executada. O **RECOVERY** é executado em paralelo com o retorno mal sucedido do resultado. Esse serviço faz com que o *atributo* seja avaliado em um escopo global em vez de escopo de função. Disponível apenas se os serviços de fluxo de trabalho estiverem ativados (consulte **#OPTION(workflow)**).

Exemplo:

```
DoSomethingToFixIt := TRUE; //some action to repair the input

SPeople := SORT(Person, Person.per_first_name);

nUniques := DEDUP(sPeople, Person.per_first_name AND Person.address)
           :RECOVERY(DoSomethingToFixIt, 2),
           FAILURE(Email.simpleSend(SystemsPersonel,
                                     SystemsPersonel.email,
                                     'ouch.htm'));
```

Ver também: **SUCCESS**, **FAILURE**

STORED - Serviço de Fluxo de Trabalho

[attribute :=] expression : STORED(storedname [, FEW], FORMAT([SELECT(valuestring)] [FIELDWIDTH(widthvalue)][,FIELDHEIGHT(heightvalue)][,SEQUENCE(sequencevalue)][,NOINPUT][,PASSWORD])]) ;

<i>attribute</i>	Opcional. O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>storedname</i>	Uma constante de string que contém o nome do resultado do atributo armazenado.
FEW	Opcional. Quando a expressão for um dataset ou um recordset, FEW especifica que o dataset é totalmente armazenado dentro da workunit. Se não especificada, o dataset será armazenado como um arquivo THOR e a workunit conterá apenas o nome do arquivo. A opção FEW é obrigatória ao usar STORED em uma MACRO com SOAP ativada e quando a entrada esperada for um dataset (tal como tns.xmlDataset).
FORMAT	Opcional. FORMAT especifica as opções de formatação do campo em um formulário Web no WsECL.
SELECT	Opcional. SELECT especifica um controle das entradas em lista suspensa no formulário Web no WsECL.
<i>valuestring</i>	Uma string que contém os valores possíveis da lista suspensa. Um asterisco (*) denota o valor padrão. Uma expressão na forma 'apple=1' dentro da string permite que o texto seja exibido e que um valor distinto seja armazenado. Neste exemplo, apple seria exibida, mas o valor 1 será armazenado se o usuário selecionar apple.
FIELDWIDTH	Opcional. FIELDWIDTH especifica a largura da caixa das entradas em um formulário Web no WsECL.
<i>widthvalue</i>	Uma expressão de valor inteiro que define a largura (o número de caracteres) da caixa das entradas.
FIELDHEIGHT	Opcional. FIELDHEIGHT especifica a altura da caixa de entradas em um formulário Web no WsECL.
<i>heightvalue</i>	Uma expressão de valor inteiro que define a altura (o número de linhas) da caixa das entradas.
SEQUENCE	Opcional. SEQUENCE especifica a ordenação do campo em um formulário Web no WsECL.
<i>sequencevalue</i>	Uma expressão de valor inteiro que define a localização sequencial da caixa das entradas. Estes valores podem ser esparsos (p.ex., 100, 200, 300) para permitir a inserção de novas entradas futuramente.
NOINPUT	Opcional. Se NOINPUT for especificado, o campo não será exibido no formulário Web no WsECL.
PASSWORD	Opcional. Se PASSWORD for especificado, uma caixa de entradas de senha será usada no formulário Web no WsECL, e o valor inserido no campo não será mostrado enquanto estiver sendo inserido. O valor também será ocultado ao visualizar os valores armazenados na tarefa através do ECLWatch ou a partir da linha de comando ao extrair a WU XML.

O serviço **STORED** armazena o resultado da *expressão* com a workunit que usa o *atributo* para que permaneça disponível para uso em toda a workunit. Se o nome do *atributo* for omitido, o valor armazenado só pode ser acessado mais tarde, quando ECL não estiver sendo executado. Se o nome do *atributo* for fornecido, o valor deste *atributo* será extraído do armazenamento; se o nome ainda não tiver sido definido, ele será computado, armazenado e então extraído do armazenamento. Este serviço implicitamente faz com que o *atributo* seja avaliado em um escopo global em vez de um escopo isolado.

STORED cria um espaço de armazenamento na tarefa onde a interface pode colocar os valores que serão especificados para uma consulta publicada. Consulte a seção *Trabalhando com Roxie* no *Guia do Programador*.

Exemplo:

```
COUNT(person) : STORED('myname');
// Name in work unit is myname,
// stored value accessible only outside ECL
fred := COUNT(person) : STORED('fred');
// Name in work unit is fred
fred := COUNT(person) : STORED('mindy');
// Name in work unit is mindy

//FORMAT options for WsECL form

Password := '' := STORED('Password',FORMAT(SEQUENCE(1),PASSWORD));
//password entry box on form
Field1 := 1 : STORED('Field1',FORMAT(SEQUENCE(10)));
Field2 := 2 : STORED('Field2',FORMAT(SEQUENCE(20)));
AddThem := TRUE :STORED ('AddThem',FORMAT(SEQUENCE(15)));
// places field in between Field1 and Field2
HiddenValue := 12 :STORED ('HiddenValue',FORMAT(NOINPUT)); // not on form
TextField1 := 'Fill in description' :Stored('Description',
FORMAT(FIELDWIDTH(25),FIELDHEIGHT(2),
SEQUENCE(5)));
//Creates 25 char wide, 2 row high input box

//SELECT options

UNSIGNED8 u8 := 0 : STORED('u8', FORMAT(fieldwidth(8),
SEQUENCE(18),
SELECT('one=1,two=2,three=3,*four=4')));
STRING ch1 := 'ban' : STORED('ch1', FORMAT(SELECT('apple=app,pear,*banana=ban')));
//banana is default
STRING ch2 := '' : STORED('ch2', FORMAT(SELECT('apple=app,pear,banana=ban')));
//starts empty, no specified default
STRING ch3 := '' : STORED('ch3', FORMAT(SELECT('apple=app,pear*,banana=ban')));
//empty in middle, empty is default
```

Ver também: Função STORED,, #WEBSERVICE

SUCCESS

attribute := *expression* : **SUCCESS**(*handler*) ;

<i>attribute</i>	O nome do atributo.
<i>expression</i>	A definição do atributo.
<i>handler</i>	A ação a ser executada se a expressão for bem-sucedida.

O serviço **SUCCESS** executa a o atributo handler quando a expressão é bem sucedida. O **SUCCESS** é executado em paralelo com o retorno bem-sucedido do resultado. Esse serviço implicitamente faz com que o *atributo* seja avaliado em um escopo global em vez de escopo de função. Disponível apenas se os serviços de fluxo de trabalho estiverem ativados (consulte #OPTION(workflow)).

Exemplo:

```
SPeople := SORT(Person, Person.first_name);
nUniques := COUNT(DEDUP(sPeople, Person.per_first_name AND
                        Person.address))
           : SUCCESS(Email.simpleSend(SystemsPersonel,
                                        SystemsPersonel.email, 'yeah.htm'));
```

Ver também: FAILURE, RECOVERY

WHEN

action : **WHEN**(*event* [, **COUNT**(*repeat*)]);

<i>action</i>	Qualquer ação ECL válida a ser executada.
<i>event</i>	O evento disparador da execução da ação. Pode ser as funções EVENT ou CRON , EVENTNAME ou o nome de um EVENTO EVENT (como um atalho para EVENT(event, '*')), ou qualquer atributo definido com essas funções.
<i>COUNT</i>	Opcional. Especifica o número de eventos que acionarão as instâncias da ação. Se omitido, o padrão é ilimitado (continuamente aguardando outro evento acionar outra instância da ação), até que a tarefa seja manualmente removida da lista das tarefas que estão sendo monitoradas pelo agendador.
<i>repeat</i>	Uma expressão de valor inteiro.

O serviço **WHEN** executa a *ação* sempre que o *evento* for acionado.

Exemplo:

```
IMPORT STD;
IF (STD.File.FileExists('test::myfile'),
    STD.File.DeleteLogicalFile('test::myfile'));
    //deletes the file if it already exists
STD.File.MonitorLogicalFileName('MyFileEvent', 'test::myfile');
    //sets up monitoring and the event name
    //to fire when the file is found
OUTPUT('File Created') : WHEN(EVENT('MyFileEvent', '*'));
    //this OUTPUT occurs only after the event has fired
    //may also be coded in this shorthand form:
    // OUTPUT('File Created') : WHEN('MyFileEvent');
afile := DATASET([{'A', '0'}], {STRING10 key, STRING10 val});
OUTPUT(afile, 'test::myfile');
    //this creates a file that the DFU file monitor will find
    //when it periodically polls
    //*****
EXPORT events := MODULE
    EXPORT dailyAtMidnight := CRON('0 0 * * *');
    EXPORT dailyAt( INTEGER hour,
        INTEGER minute=0 ) :=
        EVENT('CRON',
            (STRING)minute + ' ' + (STRING)hour + ' * * *');
    EXPORT dailyAtMidday := dailyAt(12, 0);
END;
BUILD(teenagers) : WHEN(events.dailyAtMidnight);
BUILD(oldies) : WHEN(events.dailyAt(6));
BUILD(oldies) : WHEN(EVENT('FileDropped', 'x'));
```

Ver também: **EVENT**, **CRON**, **NOTIFY**, **WAIT**

Linguagem Template

Visão Geral da Linguagem Template

A ECL foi criada para ser a linguagem de programação de toda a nossa tecnologia HPCC. Portanto, a linguagem deve ser capaz de atender a todas as demandas de uma solução de negócios completa, do consumo, query e processamento de dados até o atendimento e a saída para o cliente.

Na maioria das soluções de negócios que criamos, os usuários finais usam algum tipo de aplicativo de interface gráfica do usuário (GUI) específica dos negócios (normalmente, criada por nós) para especificar queries de dados e configurar o processamento de tarefas para o supercomputador. Esses aplicativos de GUI personalizados podem gerar para os usuários a ECL que realmente executará a query ou o processo. A tarefa de gerar essa ECL pode ser um grande desafio do ponto de vista de codificação, quando consideramos a curva exponencial de todos os possíveis conjuntos de escolhas que o usuário pode fazer em um sistema moderadamente complexo. À medida que o sistema se torna mais complexo, o problema fica se agrava ainda mais. Isso significa que uma solução de codificação é inviável.

A linguagem template da ECL oferece a solução para esse problema. A linguagem template é uma metalinguagem que recebe entradas em XML padrão, normalmente geradas por um aplicativo de GUI do usuário final (simplificando consideravelmente, dessa forma, o problema de codificação na GUI) e gera o código ECL adequado para implementar as escolhas do usuário.

Declarações de Linguagem Template

Todas as declarações da linguagem template começam por # para diferenciá-los claramente do código ECL que será gerado pelo modelo. A maioria das declarações recebem parâmetros que determinam sua ação específica em cada instância.

O terminador de declaração obrigatório é o ponto e vírgula (assim como na ECL) e há estruturas de várias linhas que terminam com a declaração #END. Essas estruturas podem estar aninhadas uma dentro da outra.

Símbolos Template

A linguagem template usa símbolos definidos pelo usuário como variáveis. Esses símbolos devem ser declarados explicitamente antes do uso (consulte #DECLARE). **Os nomes de tag no texto XML sendo processado também são tratados como símbolos definidos pelo usuário.**

Para fazer referência a um símbolo definido pelo usuário ou a uma tag XML, o nome do símbolo ou tag deve estar entre sinais de porcentagem. Uma tag XML usada como *symbol* de template pode ser um simples nome de tag ou um xpath para os dados XML a serem recuperados (consulte a documentação da estrutura RECORD para obter uma descrição da sintaxe de xpath permitida). Se um xpath for usado, o *symbol* usado deve ser o xpath completo para os dados, especificado entre chaves ({}). A sintaxe pode assumir várias formas:

%symbol%	retorna o valor do símbolo
%'symbol'%.	retorna o valor do símbolo como string
%" %	(uma string vazia) retorna o conteúdo da tag XML atual
%{xpath}%	retorna o valor dos dados apontados pelo xpath
%'{xpath}'%.	retorna o valor dos dados apontados pelo xpath como uma string

#APPEND

#APPEND(*symbol*, *expression*);

<i>symbol</i>	O nome de um símbolo definido pelo usuário previamente declarado.
<i>expression</i>	A expressão de string que especifica a string a ser concatenada ao conteúdo dos símbolos existentes.

A declaração **#APPEND** adiciona o valor da *expressão* ao final do conteúdo da string existente do *símbolo*.

Exemplo:

```
#DECLARE(MySymbol);           //declare a symbol named "MySymbol"  
#SET(MySymbol,'Hello');       //initialize MySymbol to "Hello"  
#APPEND(MySymbol,' World');   //make MySymbol's value "Hello World"
```

Ver também: #DECLARE, #SET

#CONSTANT

#CONSTANT(*name*, *value*);

<i>name</i>	Uma constante de string que contém o nome do valor armazenado.
<i>value</i>	Uma expressão para o valor a ser atribuído ao nome armazenado.

A instrução **#CONSTANT** é semelhante a **#STORED** porque atribui o *valor* ao *nome*, mas **#CONSTANT** especifica que o valor não pode ser sobregravado no tempo de execução. Essa instrução pode ser usada fora de um escopo XML e não requer um **LOADXML** anterior para instanciar um escopo XML.

Exemplo:

```
PersonCount := 0 : STORED('myname');  
#CONSTANT('myname',100);  
//make stored PersonCount attribute value to 100
```

Ver também: **STORED**

#DECLARE

#DECLARE(*symbol*);

<i>symbol</i>	O nome da variável do modelo.
---------------	-------------------------------

A declaração **#DECLARE** declara um *symbol* definido pelo usuário para uso no modelo. O *symbol* é simplesmente criado e não é inicializado para nenhum valor específico. Portanto, pode ser destinado a conter dados string ou numéricos.

Exemplo:

```
#DECLARE(MySymbol); //declare a symbol named "MySymbol"  
#SET(MySymbol,1); //initialize MySymbol to 1
```

Ver também: #SET, #APPEND

#DEMANGLE

#DEMANGLE(*identifier*);

<i>identifier</i>	Um rótulo identificador de ECL válido contendo apenas caracteres de letras, números, cifrão (\$) e sublinhado (_).
-------------------	--

A **#DEMANGLE** statements *identifier* a declaração **#DEMANGLE** usa uma string de identificador e retorna a string como era antes de passar pelo **#MANGLED**.

Exemplo:

```
#DECLARE (mstg);
#DECLARE (dmstg);
#SET (mstg, #MANGLE('SECTION_STATES/AREACODES'));

export res1 := %'mstg'%;
res1;      //res1 = 'SECTION_5fSTATES_2fAREACODES'

// Do some processing with ECL Valid Label name "mstg"

#SET (dmstg, #DEMANGLE(%'mstg'%));
export res2 := %'dmstg'%;
res2; //res2 = 'SECTION_STATES/AREACODES'
```

Ver também: **#MANGLE**, Nomes do atributo.

#ERROR

#ERROR(*ERROR* (*errorcode* , *errormessage*) ;);

<i>ERROR</i> (<i>errorcode</i> , <i>errormessage</i>);	Uma constante de string que contém a mensagem a ser exibida.
--	--

A declaração **#ERROR** interrompe imediatamente o processamento na tarefa e exibe *errormessage*. Essa declaração pode ser usada fora de um escopo XML e não exige um LOADXML anterior para instanciar um escopo XML.

Exemplo:

```
#IF ( TRUE )
  #ERROR( 'broken' );
  OUTPUT( 'broken' );
#ELSE
  #WARNING( 'maybe broken' );
  OUTPUT( 'maybe broken' );
#END;
```

Ver também: #WARNING

#EXPAND

#EXPAND(*token*);

<i>token</i>	O nome do parâmetro MACRO cujo valor de constante de string especificado será expandido.
--------------	--

A declaração **#EXPAND** substitui e analisa o texto da string do *token* especificado na MACRO.

Exemplo:

```
MAC_join(attrname, leftDS, rightDS, linkflags) := MACRO
    attrname := JOIN(leftDS,rightDS,#EXPAND(linkflags));
ENDMACRO;

MAC_join(J1,People,Property,'LEFT.ID=RIGHT.PeopleID,LEFT OUTER')
//expands out to:
// J1 := JOIN(People,Property,LEFT.ID=RIGHT.PeopleID,LEFT OUTER);

MAC_join(J2,People,Property,'LEFT.ID=RIGHT.PeopleID')
//expands out to:
// J2 := JOIN(People,Property,LEFT.ID=RIGHT.PeopleID);
```

Ver também: MACRO

#EXPORT

#EXPORT(*symbol*, *data*);

<i>symbol</i>	O nome de uma variável de modelo previamente declarada.
<i>data</i>	O nome de um campo, estrutura RECORD ou dataset.

A statement **#EXPORT** produz XML em formato texto de *dados* especificados e substitui no lugar do *symbol*. Isso permite o formato LOADXML(symbol,name) para instanciar um escopo XML com base nas informações de *dados* a serem processadas.

A saída XML é gerada no seguinte formato:

```
<Data>
  <Field label="<label-of-field>"
    name="<name-of-field>"
    position="<n>"
    rawtype="<n>"
    size="<n>"
    type="<ecl-type-without-size>" />
  ...
</Data>
```

IFBLOCKs são simplesmente expandidos no XML. Tipos RECORD aninhados têm um atributo isRecord definido como 1 e são seguidos pelos campos que contêm e por uma tag Field sem nome com o atributo isEnd definido como 1. Essa representação é usada (em vez de objetos aninhados) para que seja possível processá-la por uma declaração #FOR. Os tipos de datasets secundários também são expandidos de forma semelhante e têm um atributo isDataset definido como 1 no campo.

Exemplo:

```
NamesRecord := RECORD
  STRING10 first;
  STRING20 last;
END;
r := RECORD
  UNSIGNED4 dg_parentid;
  STRING10 dg_firstname;
  STRING dg_lastname;
  UNSIGNED1 dg_prange;
  IFBLOCK(SELF.dg_prange % 2 = 0)
    STRING20 extrafield;
  END;
  NamesRecord namerec;
  DATASET(NamesRecord) childNames;
END;

ds := DATASET('~RTTEST::OUT::ds', r, thor);

#DECLARE(out);
#EXPORT(out, r);
OUTPUT('%out%');
/* produces this result:
<Data>
  <Field label="DG_ParentID"
    name="DG_ParentID"
    position="0"
    rawtype="262401"
    size="4"
    type="unsigned integer"/>
  <Field label="DG_firstname"
```

```
    name="DG_firstname"
    position="1"
    rawtype="655364"
    size="10"
    type="string"/>
<Field label="DG_lastname"
    name="DG_lastname"
    position="2"
    rawtype="-983036"
    size="-15"
    type="string"/>
<Field label="DG_Prange"
    name="DG_Prange"
    position="3"
    rawtype="65793"
    size="1"
    type="unsigned integer"/>
<Field label="ExtraField"
    name="ExtraField"
    position="4"
    rawtype="1310724"
    size="20"
    type="string"/>
<Field isRecord="1"
    label="namerec"
    name="namerec"
    position="5"
    rawtype="13"
    size="30"
    type="namesRecord"/>
<Field label="first"
    name="first"
    position="6"
    rawtype="655364"
    size="10"
    type="string"/>
<Field label="last"
    name="last"
    position="7"
    rawtype="1310724"
    size="20"
    type="string"/>
<Field isEnd="1" name="namerec"/>
<Field isDataset="1"
    label="childNames"
    name="childNames"
    position="8"
    rawtype="-983020"
    size="30"
    type="table of &lt;unnamed&gt;"/>
<Field label="first"
    name="first"
    position="9"
    rawtype="655364"
    size="10"
    type="string"/>
<Field label="last"
    name="last"
    position="10"
    rawtype="1310724"
    size="20"
    type="string"/>
<Field isEnd="1" name="childNames"/>
</Data>
*/
```

```
//which you can then process ;ike this:
LOADXML('%out%', 'Fred');
#FOR (Fred)
  #FOR (Field)
    #IF (%'{@isEnd}'% <> '')
      OUTPUT('END');
    #ELSE
      OUTPUT('%'{@type}'%
        #IF (%'{@size}'% <> '-15' AND
          %'{@isRecord}'%='' AND
          %'{@isDataset}'%='')
        + %'{@size}'%
        #END
        + ' ' + %'{@label}'% + ';'');
      #END
    #END
  #END
OUTPUT('Done');
```

Ver também: `LOADXML`, `#EXPORTXML`, `#DECLARE`

#EXPORTXML

#EXPORTXML(*symbol*, *data*);

<i>symbol</i>	O nome de uma variável de modelo que não foi declarada previamente.
<i>data</i>	O nome de um campo, estrutura RECORD ou dataset.

A declaração **#EXPORTXML** produz o mesmo XML que **#EXPORT** dos *dados* especificados e a coloca no *símbolo*, em seguida, faz um LOADXML (*símbolo*, 'label') nos dados.

Exemplo:

```
NamesRecord := RECORD
    STRING10 first;
    STRING20 last;
END;

r := RECORD
    UNSIGNED4 dg_parentid;
    STRING10 dg_firstname;
    STRING dg_lastname;
    UNSIGNED1 dg_prange;
    IFBLOCK(SELF.dg_prange % 2 = 0)
        STRING20 extrafield;
    END;
    NamesRecord namerec;
    DATASET(NamesRecord) childNames;
END;

ds := DATASET('~RTTEST::OUT::ds', r, THOR);

//This example produces the same result as the example for #EXPORT.
//Notice the lack of #DECLARE and LOADXML in this version:
#EXPORTXML(Fred,r);

#FOR (Fred)
    #FOR (Field)
        #IF (%{@isEnd}'% <> '')
            OUTPUT('END');
        #ELSE
            OUTPUT(%{@type}'%
                #IF (%{@size}'% <> '-15' AND
                    %{@isRecord}'%='' AND
                    %{@isDataset}'%='')
                + %{@size}'%
                #END
                + ' ' + %{@label}'% + ';');
            #END
        #END
    #END
OUTPUT('Done');
//*****
//These examples show some other possible uses of #EXPORTXML:

//This could be greatly simplified as
// (%{@IsAStringMetaInfo/Field[1]/@type}'%='string')
isAString(inputField) := MACRO
#EXPORTXML(IsAStringMetaInfo, inputField);
#IF (%'IsAString'%='')
    #DECLARE(IsAString);
#END;
```



```
#SET(IsAString, false);
#FOR (IsAStringMetaInfo)
  #FOR (Field)
    #IF (%{@type}% = 'string')
      #SET (IsAString, true);
    #END
  #BREAK
#END
#END
%IsAString%
ENDMACRO;

getFieldName(inputField) := MACRO
  #EXPORTXML(GetFieldNameMetaInfo, inputField);
  %'{GetFieldNameMetaInfo/Field[1]/@name}'%
ENDMACRO;

displayIsAString(inputField) := MACRO
  OUTPUT(getFieldName(inputField)
    + TRIM(IF(isAString(inputField), ' is', ' is not'))
    + ' a string.')
ENDMACRO;

SIZEOF(r.dg_firstname);
isAString(r.dg_firstname);
getFieldName(r.dg_firstname);
OUTPUT('ds.dg_firstname isAString? '
  + (STRING)isAString(ds.dg_firstname));
isAString(ds.namerec);

displayIsAString(ds.namerec);
displayIsAString(r.dg_firstname);
```

Ver também: LOADXML, #EXPORT

#FOR

#FOR(*tag* [(*filter*)])

statements

#END

<i>tag</i>	Uma tag XML.
<i>filter</i>	Uma expressão lógica que indica quais instâncias de tag específicas serão processadas.
<i>statements</i>	As declarações de modelos a serem executadas.
#END	O terminador de estrutura #FOR .

A estrutura **#FOR** executa loops pelo XML, procurando cada instâncias de *tag* que cumpre a expressão *filter* e executa *statements* nos dados contidos nessa *tag*.

Exemplo:

```
// This script processes XML and generates ECL COUNT statements
// which run against the datasets and filters specified in the XML.
XMLstuff :=
  '<section>'+
    '<item>'+
      '<dataset>person</dataset>'+
      '<filter>firstname = \'RICHARD\'</filter>'+
    '</item>'+
    '<item>'+
      '<dataset>person</dataset>'+
      '<filter>firstname = \'JOHN\'</filter>'+
    '</item>'+
    '<item>'+
      '<dataset>person</dataset>'+
      '<filter>firstname = \'HENRY\'</filter>'+
    '</item>'+
  '</section>';

LOADXML(XMLstuff);
#DECLARE(CountStr); // Declare CountStr
#SET(CountStr, ' '); // Initialize it to an empty string
#FOR(item)
  #APPEND(CountStr, 'COUNT(' + '%dataset%' + '(' + '%filter%' + ' ));\n');
#END

OUTPUT('%CountStr%'); // output the string just built
%CountStr% // then execute the generated "COUNT" actions

// Note that the "CountStr" will have 3 COUNT actions in it:
//   COUNT(person(person.firstname = 'RICHARD'));
//   COUNT(person(person.firstname = 'JOHN'));
//   COUNT(person(person.firstname = 'HENRY'));
```

Ver também: **#LOOP**, **#DECLARE**

#GETDATATYPE

#GETDATATYPE(*field*);

<i>field</i>	Um símbolo previamente definido pelo usuário contendo o nome de um campo em um dataset.
--------------	---

A função **#GETDATATYPE** retorna o tipo de valor de *field*.

Exemplo:

```
person := DATASET([{'D'6789ABCDE6789ABCDE'}], {DATA9 per_cid});
#DECLARE(fieldtype);
#DECLARE(field);
#SET(field, 'person.per_cid');
#SET(fieldtype, #GETDATATYPE(%field%));
res := '%fieldtype%';
res; // Output: res = 'data9'
```

Ver também: Tipos de valores

#IF

#IF(*condition*)

truestatements

[**#ELSEIF**(*condition*)

truestatements]

[**#ELSE** *falsestatements*]

#END

<i>condition</i>	Uma expressão lógica.
<i>truestatements</i>	As declarações de modelos a serem executadas se a condição for verdadeira.
#ELSEIF	Opcional. Fornece a estrutura as declarações a serem executadas se a condição for verdadeira.
#ELSE	Opcional. Fornece as declarações a serem executadas se a condição for falsa.
<i>falsestatements</i>	Opcional. As declarações de modelos a serem executadas se a condição for falsa.
#END	O terminador da estrutura #IF .

A estrutura **#IF** avalia a *condição* executa *truestatements* ou *falsestatements* (se presentes). Essa declaração pode ser usada fora de um escopo XML e não exige um **LOADXML** anterior para instanciar um escopo XML.

Exemplo:

```
// This script creates a set attribute definition of the 1st 10
// natural numbers and defines an attribute named "Set10"

#DECLARE (SetString);
#DECLARE (Ndx);
#SET (SetString, ''); //initialize SetString to [
#SET (Ndx, 1);        //initialize Ndx to 1
#LOOP
  #IF (%Ndx% > 9)      //if we've iterated 9 times
    #BREAK            // break out of the loop
  #ELSE                //otherwise
    #APPEND (SetString, '%Ndx%' + ',');
                      //append Ndx and comma to SetString
  #SET (Ndx, %Ndx% + 1);
                      //and increment the value of Ndx
#END
#END

#APPEND (SetString, '%Ndx%' + ']'); //add 10th element and closing ]

EXPORT Set10 := '%SetString%'; //generate the ECL code
                      // This generates:
                      // EXPORT Set10 := [1,2,3,4,5,6,7,8,9,10];
```

Ver também: **#LOOP**, **#DECLARE**

#INMODULE

#INMODULE(*module*, *attribute*);

<i>module</i>	Um símbolo previamente definido pelo usuário contendo o nome de um módulo de fonte ECL.
<i>attribute</i>	Um símbolo previamente definido pelo usuário contendo o nome de um atributo que pode ou não estar no módulo.

A declaração **#INMODULE** retorna um TRUE ou FALSE booleano que informa se o *attribute* existe no *module* especificado.

Exemplo:

```
#DECLARE (mod)
#DECLARE (attr)
#DECLARE (stg)

#SET(mod, 'default')
#SET(attr, 'YearOf')

#IF( #INMODULE(%mod%, %attr%) )
    #SET(stg, '%attr%' + ' Exists In Module ' + '%mod%');
#ELSE
    #SET(stg, '%attr%' + ' Does Not Exist In Module ' + '%mod%');
#END

export res := '%stg%';
res;

// Output: (For 'default.YearOf')
// stg = 'YearOf Exists In Module default'
//
// Output: (For 'default.Fred')
// stg = 'Fred Does Not Exist In Module default'
```

#LOOP / #BREAK

#LOOP

[*statements*]

#BREAK

[*statements*]

#END

<i>statements</i>	As declarações de modelos a serem executadas a cada vez.
#BREAK	Encerra o loop.
#END	O terminador da estrutura #LOOP .

A estrutura **#LOOP** itera executando *statements* a cada passagem pelo loop até que uma declaração **#BREAK** seja executada. Se não ocorrer um **#BREAK**, o **#LOOP** iterará infinitamente.

Exemplo:

```
// This script creates a set attribute definition of the 1st 10
// natural numbers and defines an attribute named "Set10"

#DECLARE (SetString)
#DECLARE (Ndx)
#SET (SetString, ''); //initialize SetString to [
#SET (Ndx, 1); //initialize Ndx to 1
#LOOP
  #IF (%Ndx% > 9) //if we've iterated 9 times
    #BREAK // break out of the loop
  #ELSE //otherwise
    #APPEND (SetString, '%Ndx%' + ',');
    //append Ndx and comma to SetString
  #SET (Ndx, %Ndx% + 1)
    //and increment the value of Ndx
  #END
#END

#APPEND (SetString, '%Ndx%' + ']'); //add 10th element and closing ]

EXPORT Set10 := '%SetString%'; //generate the ECL code
// This generates:
// EXPORT Set10 := [1,2,3,4,5,6,7,8,9,10];
```

Ver também: **#FOR**, **#DECLARE**, **#IF**

#MANGLE

#MANGLE(*string*);

<i>string</i>	Um valor de string.
---------------	---------------------

A declaração **#MANGLE** recebe uma string como parâmetro de entrada e retorna um rótulo de identificador ECL válido contendo apenas letras, números e caracter sublinhado (_). **#MANGLE** substitui caracteres não alfanuméricos por um caractere de sublinhado (_), seguido pelo valor hexa do caractere que está sendo substituído.

Exemplo:

```
#DECLARE (mstg)
#DECLARE (dmstg)

#SET (mstg, #MANGLE('SECTION_STATES/AREACODES'));
export res1 := '%mstg%';
res1;          //res1 = 'SECTION_5fSTATES_2fAREACODES'

// Do some processing with ECL Valid Label name "mstg"

#SET (dmstg, #DEMANGLE('%mstg%'));
export res2 := '%dmstg%';
res2;          //res2 = 'SECTION_STATES/AREACODES'
```

Ver também: **#DEMANGLE**, Nomes do Atributo

#ONWARNING

#ONWARNING(*code*, *action*);

<i>code</i>	O número exibido na coluna “Code” da caixa de ferramentas de erros de sintaxe do ECL IDE.
<i>action</i>	Uma destas ações: ignorar, erro ou aviso.

A declaração **#ONWARNING** permite especificar globalmente como processar avisos específicos. Você pode tratar a declaração como aviso, promovê-la a um erro ou ignorá-la. Avisos úteis podem ficar perdidos em um grande número de outros menos úteis. Esse recurso permite limpar a "confusão".

O serviço de fluxo de trabalho de ONWARNING substitui todos os tratamentos globais de avisos especificados por #ONWARNING.

Exemplo:

```
#ONWARNING(1041, error);
//globally promote "Record doesn't have an explicit
// maximum record size" warnings to errors
rec := { STRING x } : ONWARNING(1041, ignore);
//ignore "Record doesn't have an explicit maximum
// record size" warning on this attribute, only
```

Ver também: ONWARNING

#OPTION

#OPTION(*option*, *value*);

<i>option</i>	Uma constante de string que faz distinção entre maiúsculas e minúsculas e contém o nome da opção a ser definida.
<i>value</i>	O valor a ser definido para a opção. Pode ser qualquer tipo de valor, dependendo do tipo esperado pela opção.

Normalmente, a declaração **#OPTION** é uma diretiva de compilador que indica ao gerador de código a melhor forma de gerar o código executável de uma workunit. Essa declaração pode ser usada fora de um escopo XML e não exige uma chamada anterior à função LOADXML para instanciar um escopo XML.

Definição de Termos

Essas definições são termos "apenas para uso interno" usados nas definições de *option* a seguir.

<i>DFA</i>	Deterministic Finite-state Automaton (Autômato determinístico de estado finito.)
<i>Fold</i>	Converter uma expressão complexa em uma expressão equivalente mais simples. Por exemplo, a expressão "1+1" pode ser substituída por "2" sem alterar o resultado.
<i>Spill</i>	Gravação de resultados intermediários em disco para disponibilizar a memória para as etapas seguintes.
<i>Funnel</i>	O operador + (anexar arquivo) entre datasets pode ser visualizado como derramar todos os registros em um funil e obter um fluxo único de registros na saída, daí o termo "funil (funnel)".
<i>TopN</i>	Uma atividade gerada internamente usada em vez de CHOOSE(SORT(xx), n) quando n é um valor pequeno. Essa atividade pode ser calculada com eficiência muito maior que a classificação de todo o recordset e o descarte de todos eles, exceto os primeiros n.
<i>Activity</i>	Um operador da ECL que aceita um ou mais datasets como entrada.
<i>Gráficos</i>	Todas as Atividades em uma query.
<i>Subgraph</i>	Uma coleção de Atividades que podem estar todas ativas ao mesmo tempo no Thor.
<i>Peephole</i>	Um método de otimização de código que examina uma pequena quantidade de código não otimizado por vez para combinar as operações em operações mais eficientes.

Opções Disponíveis

As opções a seguir têm utilidade geral:

<i>maxRunTime</i>	Padrão: nenhum	Define o número máximo de segundos de execução de um job antes de esgotar o tempo limite
<i>freezePersists</i>	Padrão: false	Se verdadeiro, não calcular/recalcula PERSISTed
<i>expirePersists</i>	Padrão: true	Se verdadeiro, PERSISTs expirar após o período especificado. Isso é definido na configuração do Sasha (PersistExpiryDefault) ou usando #option ('defaultPersistExpiry', n), onde n é o número de dias.
<i>defaultPersistExpiry</i>	Padrão: nenhum	Se definida, PERSISTs expira após o número de dias especificado (ignorando a configuração PersistExpiry-Default do Sasha).
<i>multiplePersistInstances</i>	Padrão: true	Se verdadeiro, múltiplos PERSISTs são o padrão.

Referência a Linguagem ECL
Linguagem Template

<i>defaultNumPersistInstances</i>	Padrão: nenhum	Especifica o número padrão de PERSISTs. Um valor de -1 especifica que todas as cópias devem ser mantidas até expirarem ou serem excluídas manualmente.
<i>check</i>	Padrão: true	Se verdadeiro, verificar possíveis estouros de registros.
<i>expandRepeatAnyAsDfa</i>	Padrão: true	Se for verdade, expanda ANY * em um DFA.
<i>forceFakeThor</i>	Padrão: false	Se verdadeiro, forçar o código a usar hthor.
<i>forceGenerate</i>	Padrão: false	Se verdadeiro, forçar a geração de .SO, mesmo se não valer a pena
<i>globalFold</i>	Padrão: true	Se verdadeiro, executar um constant folding (resolução das expressões que podem ser calculadas durante a compilação) global antes da geração .
<i>globalOptimize</i>	Padrão: false	Se verdadeiro, executar uma otimização global.
<i>groupAllDistribute</i>	Padrão: false	Se for verdade, GROUP, ALL gerará um DISTRIBUTE em vez de um SORT global.
<i>maximizeLexer</i>	Padrão: false	Se verdadeiro, maximizar a quantidade de trabalho executada no analisador léxico.
<i>maxLength</i>	Padrão: 4096	Especificar o comprimento máximo de um registro.
<i>minimizeSpillSize</i>	Padrão: false	Se verdadeiro, se um spill for filtrado/deduplicado etc. durante a leitura, reduzir o tamanho do arquivo de spill dividindo e filtrando antes de gravar.
<i>optimizeGraph</i>	Padrão: true	Se verdadeiro, otimizar expressões em um gráfico antes da geração
<i>orderDiskFunnel</i>	Padrão: true	Se verdadeiro, se todas as entradas de um funil forem leituras de disco, executar um pull in
<i>parseDfaComplexity</i>	Padrão: 2000	Complexidade máxima de uma expressão a ser convertida em um DFA.
<i>pickBestEngine</i>	Padrão: true	Se verdadeiro, usar hthor se for mais eficiente que o Thor
<i>targetClusterType</i>	hthor Thor roxie	O tipo de supercomputador para o qual o código está sendo gerado.
<i>topnLimit</i>	Padrão: 10000	Número máximo de registros para execução de topN.
<i>outputLimit</i>	Padrão: 10	Define o tamanho máximo (em Mb) do resultado armazenado na workunit.
<i>sortIndexPayload</i>	Padrão: true	Especifica a classificação (ou não) de campos da carga útil
<i>workflow</i>	Padrão: true	Especifica a habilitação/desabilitação de serviços de fluxo de trabalho.
<i>foldstored</i>	Padrão: false	Especifica que todas as variáveis armazenadas são substituídas por seus valores padrão ou pelos valores substituídos por #stored. Isso pode reduzir consideravelmente o tamanho do gráfico gerado.
<i>skipFileFormatCrcCheck</i>	Padrão: false	Especifica que a verificação de CRC nos índices gera um aviso e não um erro.
<i>allowedClusters</i>	Padrão: nenhum	Especifica a lista delimitada por vírgulas dos nomes dos clusters (como uma constante de string) onde a worku-

Referência a Linguagem ECL
Linguagem Template

		nit pode ser executada. Isso permite que o job seja alternado entre clusters, de forma manual ou automática, caso o job seja bloqueado em seu cluster atribuído e um outro cluster válido esteja disponível para uso.
<i>AllowAutoQueueSwitch</i>	Padrão: false	Se verdadeiro, especifica que a workunit é reatribuída automaticamente para execução em outro cluster disponível listado em allowedClusters quando a bloqueada em seu cluster atribuído.
<i>performWorkflowCse</i>	Padrão: false	Se verdadeiro, especifica que o gerador de código detecta automaticamente oportunidades eliminação de subexpressões comuns que podem estar "ocultas" em vários atributos PERSISTED. Se falso, as notificações dessas oportunidades são exibidas para o programador como sugestões para o uso do serviço de fluxo de trabalho INDEPENDENT.
<i>defaultSkewError</i>	Padrão: nenhum	Um valor entre 0,0 e 1,0 que determina a quantidade de distorção necessária para gerar um erro de distorção. Esse valor é ignorado se a ECL forneceu um atributo SKEW.
<i>defaultSkewWarning</i>	Padrão: nenhum	Um valor entre 0,0 e 1,0 que determina a quantidade de distorção necessária para gerar um aviso de distorção. Se definida com um valor superior a defaultSkewError, o valor é ignorado.
<i>overrideSkewError</i>	Padrão: nenhum	Se definida como um valor entre 0,0 e 1,0, substitui todos os valores de atributos SKEW(nn) do ECL no job atual.
<i>defaultSkewThreshold</i>	Padrão: 1GB	O tamanho do dataset (em bytes) local em um único nó necessário antes que erros/avisos de Distorção sejam gerados se nenhum THRESHOLD(nn) foi fornecido na ECL.
<i>overrideSkewThreshold</i>	Padrão: nenhum	O tamanho do dataset (em bytes) local em um único nó necessário antes que erros/avisos de Distorção sejam gerados. Substitui todos os valores do atributo THRESHOLD(nn) do ECL na job atual.
<i>applyInstantEclTransformations</i>	Padrão false	Limitar saídas não destinadas a arquivos com um CHOSEN
<i>applyInstantEclTransformationsLimit</i>	Padrão é 100	Número limite de registros
<i>divideByZero</i>	Padrão zero	"zero" é avaliado como 0, o comportamento padrão. "fail" faz com que a tarefa falhe e informe um erro de divisão por zero. "nan" (no momento, permitido apenas para números reais) cria um NaN quieto, que será propagado por todas as expressões reais onde for usado. Você pode usar NOT ISVALID(x) para testar se o valor é NaN. A divisão de inteiros e decimais por zero continua a retornar 0.
<i>outputLimitMb</i>	Padrão 10 [MB]	Limite de saída para uma workunit em MB.

Referência a Linguagem ECL
Linguagem Template

<i>hthorMemoryLimit</i>	Padrão 300 [MB]	Substituir o limite de uso de memória definido na opção de configuração defaultMemoryLimitMB do ECL Agent (somente para hThor).
<i>maxCsvRowSizeMb</i>	Padrão 10 [MB]	Limite superior de leitura de uma linha CSV em MB.
<i>compressInternalSpills</i>	Padrão True	(Por exemplo, spills criados por lookahead ou coleta de classificação).
<i>hdCompressorType</i>	Padrão 'FLZ'	Distribuir o compactador a ser usado.
<i>hdCompressorOptions</i>	Padrão "	Distribuir opções do compactador (por exemplo, chave AES)
<i>splitterSpill</i>	Padrão -1	Valor inteiro para indicar se os divisores serão forçados a executar spill ou não. [1 = forçar spill 0 = forçar na memória -1 = usar a configuração do ajudante]
<i>loopMaxEmpty</i>	Padrão 1000	Número máximo de iterações de LOOP sem resultado antes de informar um erro
<i>smallSortThreshold</i>	O padrão é 0 (desabilitado).	Se o tamanho estimado for inferior ao limite em bytes, uma abordagem de miniclassificação deve ser usada.
<i>sort_max_deviance</i>	Padrão 10 [MB]	Variação máxima (bytes) permitida durante o particionamento da classificação
<i>joinHelperThreads</i>	Padrão: = igual ao número de núcleos	Número de linhas de execução (threads) a serem usadas no tipo threaded do ajudante de combinação
<i>bindCores</i>	Padrão = 0	Para consultas Roxie Se diferente de zero, configura a query para usar apenas o número especificado de núcleos. Essa definição substitui o valor coresPerQuery definido para na configuração do Roxie.
<i>translateDFSLayouts</i>	Padrão = 0	Especifica que o layout de arquivo deve ser consultado no tempo de compilação. Consulte <i>Resolução de layout de arquivo no tempo de compilação</i> no <i>Guia do Programador</i> para obter mais detalhes.
<i>timeLimit</i>		Para consultas Roxie. Tempo máximo de execução (em ms) para um consultas.
<i>generateGlobalId</i>	Padrão = false	Para consultas de Roxie. Quando true, gera um único GlobalId se um não é fornecido.
<i>analyzeWorkunit</i>		Substitui a configuração no ECL Agent para analisar as workunits depois que as consultas ECL são executadas (somente Thor). Isso permite que a workunit seja analisada para identificar e exibir potenciais problemas. Estes possíveis problemas são exibidos na área do ECL Watch "Warnings & Errors". A configuração global é padronizada para TRUE, mas pode ser alterado usando o Configuration Manager.

As opções a seguir tratam da geração de gráficos Lógicos em uma workunit.

Gráficos Lógicos são armazenados na workunit e visualizados no ECL Watch. Esses gráficos incluem informações sobre em que atributo/número de linha/coluna os símbolos são definidos. Os atributos exportados são representados por

Referência a Linguagem ECL

Linguagem Template

<module>.<attribute> no cabeçalho da atividade. Atributos não exportados (locais) são representados como <module>.<exported-attribute>::<non-exported-name>

<i>generateLogicalGraph</i>	Padrão: false	Se verdadeiro, gera um gráfico lógico além de todos os gráficos da workunit.
<i>generateLogicalGraphOnly</i>	Padrão: false	Se verdadeiro, gera apenas o Gráfico Lógico da workunit.
<i>logicalGraphExpandPersist</i>	Padrão: true	Se verdadeiro, expande atributos PERSISTED.
<i>logicalGraphExpandStored</i>	Padrão: false	Se verdadeiro, expande atributos STORED.
<i>logicalGraphIncludeName</i>	Padrão: true	Se verdadeiro, gera nomes de atributos no cabeçalho das caixas de atividade.
<i>logicalGraphIncludeModule</i>	Padrão: true	Se verdadeiro, gera nomes módulo.atributo no cabeçalho das caixas de atividades.
<i>logicalGraphDisplayJavadoc</i>	Padrão: true	Se verdadeiro, gera comentários no estilo de Javadoc integrados na ECL em vez do texto padrão que seria gerado (consulte http://java.sun.com/j2se/javadoc/writing-doccomments/). Os comentários no estilo Javadoc em estruturas ou atributos escalares de RECORD não serão gerados, pois não contam com uma caixa Atividade de gráfico diretamente associada.
<i>logicalGraphDisplayJavadocParameters</i>	Padrão: false	Se verdadeiro, gera informações sobre parâmetros em todos os comentários no Javadoc-style.
<i>filteredReadSpillThreshold</i>	Padrão: 2	É executado um spill para leituras de disco filtradas duplicadas mais de N vezes.
<i>foldConstantCast</i>	Padrão: true	Se verdadeiro, (cast)value é calculado no momento da geração.
<i>foldFilter</i>	Padrão: true	Se verdadeiro, é executado um constant folding (resolução das expressões que podem ser calculadas durante a compilação) para os filtros.
<i>foldAssign</i>	Padrão: true	Se verdadeiro, é executado um constant folding (resolução das expressões que podem ser calculadas durante a compilação) para TRANSFORMs.
<i>foldSQL</i>	Padrão: true	Se verdadeiro, é executado um constant folding (resolução das expressões que podem ser calculadas durante a compilação) para SQL.
<i>optimizeDiskRead</i>	Padrão: true	Se verdadeiro, incluir projeto e filtro na transformação para uma leitura de disco.
<i>optimizeSQL</i>	Padrão: false	Se verdadeiro, otimizar SQL.
<i>optimizeThorCounts</i>	Padrão: true	Se verdadeiro, converter COUNT(diskfile) em uma versão otimizada.
<i>peephole</i>	Padrão: true	Se verdadeiro, otimizar memcpy/memsets, etc.
<i>spotCSE</i>	Padrão: true	Se verdadeiro, procurar subexpressões comuns em TRANSFORMs/filtros.
<i>noteRecordSizeInGraph</i>	Padrão: true	Adiciona estimativas de tamanho do registro ao gráfico
<i>showActivitySizeInGraph</i>	Padrão: false	Mostrar estimativas do tamanho do ++ gerado no gráfico

Referência a Linguagem ECL
Linguagem Template

<i>showMetaInGraph</i>	Padrão: false	Adicionar ordens de distribuição/classificação ao gráfico
<i>showRecordCountInGraph</i>	Padrão: true	Mostra estimativas de quantidade de registros no gráfico
<i>spotTopN</i>	Padrão: true	Se verdadeiro, converter CHOOSSE(SORT()) em uma atividade topN.
<i>spotLocalMerge</i>	Padrão: false	Se verdadeiro, em JOIN local com ambos os lados classificados, gera uma fusão leve.
<i>countIndex</i>	Padrão: false	Se verdadeiro, otimizar COUNT(index) para a versão otimizada (também exige optimizeThorCounts).
<i>allowThroughSpill</i>	Padrão: true	Se verdadeiro, permitir through spills.
<i>optimizeBoolReturn</i>	Padrão: true	Se verdadeiro, aprimorar o código ao retornar BOOLEAN de uma função.
<i>optimizeSubString</i>	Padrão: true	Se verdadeiro, não alocar memória ao processar uma substring.
<i>thorKeys</i>	Padrão: true	Se verdadeiro, permitir operações INDEX no Thor.
<i>regexVersion</i>	Padrão: 0	Se definida como 1, especifica o uso da implementação anterior de expressões regulares, que podem ser mais rápidas, mas também podem exceder limites do stack.
<i>compileOptions</i>	Padrão: nenhum	Especificar opções de substituição do compilador (como /Zm1000 como duplicar o tamanho do heap do compilador para contornar um erro de estouro de heap).
<i>linkOptions</i>	Padrão: nenhum	Especificar opções de substituição do vinculador.
<i>otimizarProjetos</i>	Padrão: true	Se falso, desabilita a otimização automática de projeção/distribuição de campos.
<i>notifyOptimizedProjects</i>	Padrão: 0	Se definida como 1, informa otimizações para atributos nomeados. Se definida como 2, informa todas as otimizações.
<i>optimizeProjectsPreservePersists</i>	Padrão: false	Se verdadeiro, desabilita a otimização automática de projeção/distribuição de campos na leitura de arquivos PERSISTed. Se um arquivo PERSISTed é lido em um cluster de tamanho diferente do cluster onde foi criado, a otimização de campos projetados pode significar que não é possível recriar a ordem de distribuição/classificação.
<i>aggressiveOptimizeProjects</i>	Padrão: false	Se verdadeiro, habilita tentativas de minimização de tráfego de rede para classificações/distribuições. Normalmente, essa opção não gera benefícios relevantes, mas em alguns casos específicos pode gerar.
<i>PercolateConstants</i>	Padrão: true	Se falso, desabilita tentativas de otimizações agressivas de valor de constantes.

As opções a seguir são úteis para a depuração:

<i>clusterSize</i>	Padrão: nenhum	Substituir o número de nós no cluster (para testes)
--------------------	----------------	---

Referência a Linguagem ECL
Linguagem Template

<i>debugNlp</i>	Padrão: false	Se verdadeiro, gerar informações de depuração do processamento de NLP no arquivo .cpp.
<i>resourceMaxMemory</i>	Padrão: 400M	Quantidade máxima de memória que pode ser usada por um subgráfico.
<i>resourceMaxSockets</i>	Padrão: 2000	Número máximo de soquetes que podem ser usados por um subgráfico.
<i>resourceMaxActivities</i>	Padrão: 200	Número máximo de atividades que podem ser contidas em um subgráfico.
<i>unlimitedResources</i>	Padrão: false	Se verdadeiro, supor abundância de recursos ao atribuí-los para os gráficos.
<i>traceRowXML</i>	Padrão: false	Se verdadeiro, ativa o rastreamento em gráfico do ECL Watch. Use apenas com datasets pequenos para fins de depuração.
<i>_Probe</i>	Padrão: false	Se verdadeiro, exibir todas as linhas de resultados de conjuntos de resultados intermediários no gráfico do ECL Watch quando usada em conjunto com a opção <i>traceRowXML</i> . Use apenas com datasets pequenos para fins de depuração.
<i>debugQuery</i>	Padrão: false	Se verdadeiro, compilar a query usando configurações de depuração.
<i>optimizeLevel</i>	Padrão: 3 para roxie, mais 0	Definir o nível de otimização do compilador C++ (as otimizações podem aumentar consideravelmente a duração da compilação).
<i>checkAsserts</i>	Padrão: true	Se true, ativa a verificação de ASSERT.
<i>soapTraceLevel</i>	Padrão: 1	O nível de detalhes para relatar informações de SOAP-CALL ou HTTPCALL (defina como 0 para nenhuma, 1 para normal, 2 a 8 para obter mais detalhes)
<i>traceEnabled</i>	Padrão: FALSE	Habilita rastreamento para arquivos de log quando ações de TRACE estão presentes. Consulte TRACE.
<i>traceLimit</i>	Padrão: 10	Substitui a configuração padrão de KEEP para uma declaração TRACE para indicar quantas declarações TRACE são gravadas no arquivo de log. Consulte TRACE:.

As opções a seguir são para uso na geração avançada de código:

Essas *opções* não devem ser alteradas, a menos que você REALLY saiba o que está fazendo. Normalmente, elas são usadas internamente pelos nossos desenvolvedores para habilitar/desabilitar recurso que ainda estão sendo desenvolvidos. Ocasionalmente, a equipe do suporte técnico pode sugerir que você altere uma dessas configurações para contornar um problema. Caso contrário, as configurações padrão são recomendadas para todos os casos.

<i>filteredReadSpillThreshold</i>	Padrão: 2	É executado um spill para leituras de disco filtradas duplicadas mais de N vezes.
<i>foldConstantCast</i>	Padrão: true	Se verdadeiro, (cast)value é calculado no momento da geração.
<i>foldFilter</i>	Padrão: true	Se verdadeiro, é executado um constant folding (resolução das expressões que podem ser calculadas durante a compilação) para os filtros.

Referência a Linguagem ECL
Linguagem Template

<i>foldAssign</i>	Padrão: true	Se verdadeiro, é executado um constant folding (resolução das expressões que podem ser calculadas durante a compilação) para TRANSFORMs.
<i>foldSQL</i>	Padrão: true	Se verdadeiro, é executado um constant folding (resolução das expressões que podem ser calculadas durante a compilação) para SQL.
<i>optimizeDiskRead</i>	Padrão: true	Se verdadeiro, incluir projeto e filtro na transformação para uma leitura de disco.
<i>optimizeSQL</i>	Padrão: false	Se verdadeiro, otimizar SQL.
<i>optimizeThorCounts</i>	Padrão: true	Se verdadeiro, converter COUNT(diskfile) em uma versão otimizada.
<i>peephole</i>	Padrão: true	Se verdadeiro, otimizar memcpy/memsets, etc.
<i>spotCSE</i>	Padrão: true	Se verdadeiro, procurar subexpressões comuns em TRANSFORMs/filtros.
<i>spotTopN</i>	Padrão: true	Se verdadeiro, converter CHOSEN(SORT()) em uma atividade topN.
<i>spotLocalMerge</i>	Padrão: false	Se verdadeiro, em JOIN local com ambos os lados classificados, gera uma fusão leve.
<i>countIndex</i>	Padrão: false	Se verdadeiro, otimizar COUNT(index) para a versão otimizada (também exige optimizeThorCounts).
<i>allowThroughSpill</i>	Padrão: true	Se verdadeiro, permitir through spills.
<i>optimizeBoolReturn</i>	Padrão: true	Se verdadeiro, aprimorar o código ao retornar BOOLEAN de uma função.
<i>optimizeSubString</i>	Padrão: true	Se verdadeiro, não alocar memória ao processar uma substring.
<i>thorKeys</i>	Padrão: true	Se verdadeiro, permitir operações INDEX no Thor.
<i>regexVersion</i>	Padrão: 0	Se definida como 1, especifica o uso da implementação anterior de expressões regulares, que podem ser mais rápidas, mas também podem exceder limites do stack.
<i>compileOptions</i>	Padrão: nenhum	Especificar opções de substituição do compilador (como /Zm1000 como duplicar o tamanho do heap do compilador para contornar um erro de estouro de heap).
<i>linkOptions</i>	Padrão: nenhum	Especificar opções de substituição do vinculador.
<i>otimizarProjetos</i>	Padrão: true	Se falso, desabilita a otimização automática de projeção/distribuição de campos.
<i>notifyOptimizedProjects</i>	Padrão: 0	Se definida como 1, informa otimizações para atributos nomeados. Se definida como 2, informa todas as otimizações.
<i>optimizeProjectsPreservePersists</i>	Padrão: false	Se verdadeiro, desabilita a otimização automática de projeção/distribuição de campos na leitura de arquivos PERSISTed. Se um arquivo PERSISTed é lido em um cluster de tamanho diferente do cluster onde foi criado, a otimização de campos projetados pode significar que não é possível recriar a ordem de distribuição/classificação.

Referência a Linguagem ECL

Linguagem Template

<i>aggressiveOptimizeProjects</i>	Padrão: false	Se verdadeiro, habilita tentativas de minimização de tráfego de rede para classificações/distribuições. Normalmente, essa opção não gera benefícios relevantes, mas em alguns casos específicos pode gerar.
<i>PercolateConstants</i>	Padrão: true	Se falso, desabilita tentativas de otimizações agressivas de valor de constantes.
<i>exportDependencies</i>	Padrão: false	Gerar informações sobre a interdefinição de dependências
<i>maxCompileThreads</i>	Padrão: 4 para eclccserver e 1 para eclcc	Número de instâncias de compilador para compilar o c++
<i>reportCppWarnings</i>	Padrão: false	Informar os avisos da compilação do c++
<i>saveCppTempFiles</i>	Padrão: false	Retenir os arquivos c++ gerados
<i>spanMultipleCpp</i>	Padrão: true	Gerar uma tarefa em vários arquivos c++
<i>activitiesPerCpp</i>	Padrão: 500 para Linux ou 800 para Windows	Número de atividades em cada arquivo c++ (exige spanMultipleCpp)
<i>obfuscateOutput</i>	Padrão false	Se verdadeiro, os detalhes são removidos da tarefa gerada, incluindo código da ECL, estimativas de tamanho do registro e número de registros.

As seguintes opções são para o analisador de workunit:

<i>analyzeWorkunit</i>	Padrão: true	Se for indicado com FALSE, desabilita a análise da workunit
<i>analyzer_minInterestingTime</i>	Padrão: 1000	Analisa as atividades que excedem o tempo mínimo de execução (milissegundos)
<i>analyzer_minInterestingCost</i>	Padrão: 30000	Reporta problemas em que a penalidade de tempo exceda este valor (milissegundos)
<i>analyzer_skewThreshold</i>	Padrão: 20	Reporta problemas relacionados à distorção que excedam esse limite
<i>analyzer_minRowsPerNode</i>	Padrão: 1000	Ignore activities that have this average number of rows per nós

Exemplo:

```
#OPTION('traceRowXml', TRUE);
#OPTION('_Probe', TRUE);

my_rec := RECORD
  STRING20 lname;
  STRING20 fname;
  STRING2 age;
END;

d := DATASET([
  { 'PORTLY', 'STUART', '39' },
  { 'PORTLY', 'STACIE', '36' },
  { 'PORTLY', 'DARA', '1' },
  { 'PORTLY', 'GARRETT', '4' }], my_rec);

OUTPUT(d(d.age > '1'), {lname, fname, age});
```

```
//*****  
//This example demonstrates Logical Graphs and  
// Javadoc-style comment blocks  
#OPTION('generateLogicalGraphOnly',TRUE);  
#OPTION('logicalGraphDisplayJavadocParameters',TRUE);  
  
/**  
 * Defines a record that contains information about a person  
 */  
namesRecord :=  
    RECORD  
string20    surname;  
string10    forename;  
integer2    age := 25;  
    END;  
  
/**  
Defines a table that can be used to read the information from the file  
and then do something with it.  
*/  
namesTable := DATASET('x',namesRecord,FLAT);  
  
/**  
    Allows the name table to be filtered.  
  
    @param ages The ages that are allowed to be processed.  
        badForename Forname to avoid.  
  
    @return the filtered dataset.  
*/  
namesTable filtered(SET OF INTEGER2 ages, STRING badForename) :=  
    namesTable(age in ages, forename != badForename);  
  
OUTPUT(filtered([10,20,33], ''));
```

#SET

#SET(*symbol*, *expression*);

<i>symbol</i>	O nome de um símbolo definido pelo usuário previamente declarado.
<i>expression</i>	A expressão cujo valor será atribuído ao símbolo.

A declaração **#SET** atribui o valor de *expression* a *symbol*, substituindo todos os valores anteriores contidos no símbolo.

Exemplo:

```
#DECLARE(MySymbol); //declare a symbol named "MySymbol"  
#SET(MySymbol,1); //initialize MySymbol to 1
```

Ver também: #DECLARE, #APPEND

#STORED

#STORED(*storedname* , *value*);

<i>storedname</i>	Uma constante de string que contém o nome do resultado do atributo armazenado.
<i>value</i>	Uma expressão para o novo valor a ser atribuído ao atributo armazenado.

A declaração **#STORED** atribui *value* a *storedname*, substituindo todos os valores anteriores contidos no atributo armazenado. Essa declaração pode ser usada fora de um escopo XML e não exige um **LOADXML** anterior para instanciar um escopo XML.

Exemplo:

```
PersonCount := COUNT(person) : STORED('myname');
#STORED('myname',100);
//change stored PersonCount attribute value to 100
```

Ver também: **STORED**, **#CONSTANT**

#TEXT

#TEXT(*argument*);

<i>argument</i>	O parâmetro MACRO cujo texto será fornecido.
-----------------	--

A declaração **#TEXT** retorna o texto do *argument* especificado à MACRO. Essa declaração pode ser usada fora de um escopo XML e não exige um LOADXML anterior para instanciar um escopo XML.

Exemplo:

```
extractFields(ds, outDs, f1, f2='?') := MACRO

#UNIQUENAME(r);

%r% := RECORD
  f1 := ds.f1;
#IF (#TEXT(f2)<>'?')
  #TEXT(f2)+':';
  f2 := ds.f2;
#END
END;

outDs := TABLE(ds, %r%);
ENDMACRO;

extractFields(people, justSurname, lastname);
OUTPUT(justSurname);

extractFields(people, justName, lastname, firstname);
OUTPUT(justName);
```

Ver também: MACRO

#UNIQUENAME

#UNIQUENAME(*namevar* [,*pattern*]);

<i>namevar</i>	O rótulo da variável do modelo (sem os sinais de porcentagem) a ser usado em declarações subsequentes (com os sinais de porcentagem) que precisam do nome único gerado.
<i>pattern</i>	Opcional. Um modelo para a criação de um nome único. O modelo deve conter um sinal de dólar (\$) para indicar a posição em que um número único é gerado, e pode conter uma cerquilha (#) para incluir <i>namevar</i> . Isso é útil para situações em que #UNIQUENAME é usado para gerar nomes de arquivos e o resultado é destinado à visualização no programa ECL IDE, pois #UNIQUENAME por padrão gera identificadores que começam com dois caracteres de sublinhado (__) e a ECL IDE os trata como campos ocultos. Se omitido, o formato padrão é __#__\$.

A declaração **#UNIQUENAME** cria um identificador de ECL válido e único dentro do contexto do limite do escopo atual. Isso é particularmente útil em estruturas MACRO, pois permite que a macro seja usada várias vezes no mesmo escopo sem criar erros de nome de atributo duplicado nas definições do atributo dentro da macro. Essa declaração pode ser usada fora de um escopo XML e não exige um LOADXML anterior para instanciar um escopo XML.

Exemplo:

```
IMPORT Training_Compare;
EXPORT MAC_Compare_Result(module_name, attribute_name) := MACRO

#UNIQUENAME(compare_file);
%compare_file% := Training_Compare.File_Compare_Master;

#UNIQUENAME(layout_per_attr);
#UNIQUENAME(compare_attr, _MyField_$);
//the compare_attr fieldname is generated like: _MyField_1_
%layout_per_attr% := RECORD
    person.per_cid;
    %compare_attr% := module_name.attribute_name;
END;

#UNIQUENAME(person_attr_out);
%person_attr_out% := TABLE(person, %layout_per_attr%);

#UNIQUENAME(person_attr_out_dist);
%person_attr_out_dist% := DISTRIBUTE(%person_attr_out%,HASH(per_cid));

#UNIQUENAME(layout_match_out);
%layout_match_out% := RECORD
    data9 per_cid;
    boolean ValuesMatchFlag;
    TYPEOF(module_name.attribute_name) MyValue;
    TYPEOF(%compare_file%.attribute_name) CompareValue;
END;

#UNIQUENAME(layout_compare);
%layout_compare% := RECORD
    %compare_file%.per_cid;
    %compare_file%.attribute_name;
END;

#UNIQUENAME(compare_table);
%compare_table% := TABLE(%compare_file%, %layout_compare%);
#UNIQUENAME(compare_table_dist);
%compare_table_dist% := DISTRIBUTE(%compare_table%, HASH(per_cid));
#UNIQUENAME(compare_attr_to_field);
%layout_match_out% %compare_attr_to_field%(%person_attr_out% L,
```

```
%compare_table% R) := TRANSFORM
    SELF.ValuesMatchFlag := (L.%compare_attr% = R.attribute_name);
    SELF.MyValue := L.%compare_attr%;
    SELF.CompareValue := R.attribute_name;
    SELF := L;
END;

#UNIQUENAME(compare_out);
%compare_out% := JOIN(%person_attr_out_dist%,
    %compare_table_dist%,
    LEFT.per_cid = RIGHT.per_cid,
    %compare_attr_to_field%(LEFT, RIGHT),
    LOCAL);

#UNIQUENAME(match_out);
#UNIQUENAME(nomatch_out);
%match_out% := %compare_out%(ValuesMatchFlag);
%nomatch_out% := %compare_out%(~ValuesMatchFlag);

COUNT(%match_out%);
OUTPUT(CHOSEN(%match_out%, 50));
COUNT(%nomatch_out%);
OUTPUT(CHOSEN(%nomatch_out%, 50));

ENDMACRO;
```

Ver também: MACRO

#WARNING

#WARNING(*message*);

<i>message</i>	Uma constante da string que contém a mensagem de aviso a ser exibida.
----------------	---

A **#WARNING** o menu é exibido, *message* a declaração **#WARNING** exibe *message* na tarefa e/ou verificação de sintaxe. Essa declaração pode ser usada fora de um escopo XML e não exige um **LOADXML** anterior para instanciar um escopo XML.

Exemplo:

```
#IF ( TRUE )
  #ERROR( 'broken' );
  OUTPUT( 'broken' );
#ELSE
  #WARNING( 'maybe broken' );
  OUTPUT( 'maybe broken' );
#END;
```

Ver também: **#ERROR**

#WEBSERVICE

#WEBSERVICE([**FIELDS**(*fieldlist*),][**HELP**(*helptext*),][**DESCRIPTION**(*descriptiontext*),]);

FIELDS	O parâmetro FIELDS especifica a sequência de campos em formulários web WsECL. Essa lista é exclusiva. Se o atributo FIELDS está presente, somente os campos da lista de campos serão exibidos no formulário web em WsECL.
<i>fieldlist</i>	Uma lista separada por vírgulas de nomes de campos na ordem em que devem aparecer no formulário.
HELP	O parâmetro HELP especifica a adição de texto de ajuda ao formulário web do WsECL.
<i>helptext</i>	O texto de ajuda a ser exibido.
DESCRIPTION	O parâmetro DESCRIPTION especifica a adição de texto descritivo ao formulário web do WsECL.
<i>descriptiontext</i>	O texto de descrição a ser exibido.

A declaração **#WEBSERVICE** define opções para os parâmetros de entrada de um formulário web do WsECL para uma query publicada.

Exemplo:

```
#WEBSERVICE(FIELDS('Field1','AddThem','Field2'),  
             HELP('Enter Integer Values'),  
             DESCRIPTION('If AddThem is TRUE, this adds the two integers'));  
Field1 := 1 : Stored('Field1');  
Field2 := 2 :Stored('Field2');  
AddThem := TRUE :STORED ('AddThem');  
HiddenValue := 12 :STORED ('HiddenValue'); //not in fieldlist, won't display on WsECL form  
IF(AddThem,OUTPUT(Field1+Field2),OUTPUT('Not Added'));  
  
#WEBSERVICE(FIELDS('field1','field2','*'));//includes unspecified fields on the WsECL form
```

Ver também: STORED

#WORKUNIT

#WORKUNIT(*option*, *value*);

<i>option</i>	Uma constante de string que especifica o nome da opção a ser definida.
<i>value</i>	O valor para definir para a opção.

As configurações de **opções** da **#WORKUNIT** especifica o *valor* para a workunit atual. Essa declaração pode ser usada fora de um escopo XML e não exige uma chamada anterior à função **LOADXML** para instanciar um escopo XML.

As configurações válidas de *opção* são:

<i>cluster</i>	O parâmetro de valor é uma constante de string que contém o nome do cluster de destino no qual a workunit é executada.
<i>protect</i>	O parâmetro de valor especifica como “true” (verdadeiro) para indicar que a tarefa está protegida contra exclusão (ou “false” [falso], caso contrário).
<i>name</i>	O parâmetro de valor é uma constante de string que especifica o nome da workunit.
<i>priority</i>	O parâmetro de valor é uma constante de string que contém níveis baixo, normal ou alto para indicar o nível de prioridade de execução da workunit, ou um valor constante inteiro (e não uma string) para especificar o quão acima do nível alto a prioridade deve estar ("super alta").
<i>scope</i>	O parâmetro de valor é uma constante de string que contém o valor de escopo a ser usado para substituir o escopo padrão da workunit (o ID de usuário do indivíduo que faz o envio). Esse é um recurso de segurança de tarefa que exige um sistema compatível com LDAP.

Exemplo:

```
#WORKUNIT('cluster','400way'); //run the job on the 400-way target cluster
#WORKUNIT('protect',true);    //disallow deletion or archiving by Sasha
#WORKUNIT('name','My Job');   //name it "My Job"
#WORKUNIT('priority','high'); //run before other lower-priority jobs
#WORKUNIT('priority',10);     //run before other high-priority jobs
#WORKUNIT('scope','NewVal');  //override the default scope (on an LDAP enabled system)
```

Serviços Externos

Estrutura SERVICE

servicename := **SERVICE** [: *defaultkeywords*]

prototype : *keywordlist*;

END;

<i>servicename</i>	O nome do serviço fornecido pela estrutura SERVICE.
<i>defaultkeywords</i>	Opcional. Uma lista delimitada por vírgula das palavras-chave padrão e seus valores compartilhados por todos os protótipos no serviço externo.
<i>prototype</i>	O nome ECL e o protótipo de uma função específica.
<i>keywordlist</i>	Uma lista delimitada por vírgula das palavras-chave e seus valores que instruem o compilador ECL como acessar o serviço externo.

A estrutura **SERVICE** possibilita a criação de serviços externos para ampliar os recursos do ECL no desempenho de qualquer funcionalidade desejada. Estes serviços externos do sistema são implementados como funções exportadas em .SO(Shared Object). Um serviço de sistema da ECL .SO pode conter um ou mais serviços e (possivelmente) uma única rotina de inicialização de .SO.

Exemplo:

```
email := SERVICE
    simpleSend( STRING address,
                STRING template,
                STRING subject) : LIBRARY='ecl2cw',
                                INITFUNCTION='initEcl2Cw';
END;
MyAttr := COUNT(Trades): FAILURE(email.simpleSend('help@ln_risk.com',
                                                    'FailTemplate',
                                                    'COUNT failure'));
//An example of a SERVICE function returning a structured record
NameRecord := RECORD
    STRING5 title;
    STRING20 fname;
    STRING20 mname;
    STRING20 lname;
    STRING5 name_suffix;
    STRING3 name_score;
END;

LocalAddrCleanLib := SERVICE
NameRecord dt(CONST STRING name, CONST STRING server = 'x')
    : c,entrypoint='aclCleanPerson73',pure;
END;

MyRecord := RECORD
    UNSIGNED id;
    STRING uncleanedName;
    NameRecord Name;
END;
x := DATASET('x', MyRecord, THOR);

myRecord t(myRecord L) := TRANSFORM
    SELF.Name := LocalAddrCleanLib.dt(L.uncleanedName);
```

```
    SELF := L;
  END;
y := PROJECT(x, t(LEFT));
OUTPUT(y);

//The following two examples define the same functions:
TestServices1 := SERVICE
  member(CONST STRING src)
    : holertl,library='test',entrypoint='member',ctxmethod;
  takesContext1(CONST STRING src)
    : holertl,library='test',entrypoint='takesContext1',context;
  takesContext2()
    : holertl,library='test',entrypoint='takesContext2',context;
  STRING takesContext3()
    : holertl,library='test',entrypoint='takesContext3',context;
END;

//this form demonstrates the use of default keywords
TestServices2 := SERVICE : holertl,library='test'
  member(CONST STRING src) : entrypoint='member',ctxmethod;
  takesContext1(CONST STRING src) : entrypoint='takesContext1',context;
  takesContext2() : entrypoint='takesContext2',context;
  STRING takesContext3() : entrypoint='takesContext3',context;
END;
```

Ver também: Implementação de Serviços Externos, CONST

CONST

CONST

A palavra-chave **CONST** define que o valor especificado como parâmetro sempre será tratado como uma constante. Trata-se essencialmente de um indicador que permite ao compilador otimizar seu código de forma adequada ao declarar funções externas.

Exemplo:

```
STRING CatStrings(CONST STRING S1, CONST STRING S2) := S1 + S2;
```

Ver também: Funções (Especificação de parâmetro), Estrutura SERVICE

Implementação de Serviços Externos

Os serviços de sistema externos da ECL são implementados como funções exportadas em um .SO (Shared Object). Um serviço de sistema da ECL .SO pode conter um ou mais serviços e (possivelmente) uma única rotina de inicialização de .SO. Todas as bibliotecas de serviços de sistema devem ser codificadas como seguras para linha de execução (thread).

Todas as funções exportadas no .SO (denominadas neste documento como "pontos de entrada") devem cumprir algumas convenções de chamada e nomenclatura. Primeiro, os pontos de entrada devem usar a convenção de nomenclatura "C". Ou seja, a decoração do nome da função (como a usada pelo C++) não é permitida.

Segundo, a classe de armazenamento de `__declspec(dllexport)` e o tipo de declaração `_cdecl` devem ser declarados para aplicativos Windows/Microsoft C++. Normalmente, `SERVICE_CALL` é definido como `_declspec(dllexport)` e `SERVICE_API` é definida como `_cdecl` para Windows e deixada como nulos para Linux. Por exemplo:

```
Extern "C" __declspec(dllexport) unsigned _cdecl Countchars(const unsigned len, const char *string)
```

Note: O uso de um SERVICE externo pode estar limitado a módulos assinados. Consulte Assinatura do código no Guia do programador da ECL.

Inicialização .SO

Veja a seguir um exemplo de protótipo de uma rotina de inicialização de um serviço de sistema (.SO) da ECL:

```
extern "C" void stdcall <functionName> (IEclWorkUnit *w);
```

IEclWorkUnit é transparente para o aplicativo e pode ser declarado como Struct IEclWorkUnit ou simplesmente mencionado como void *.

Além disso, uma rotina de inicialização deve reter uma referência à sua "Workunit". Normalmente, uma variável global é usada para armazenar esse valor. Por exemplo:

```
IEclWorkUnit *workUnit;
// global variable to hold the Work Unit reference

extern "C" void SERVICE_API myInitFunction (IEclWorkUnit *w)
{
    workUnit = w; // retain reference to "Work Unit"
}
```

Pontos de Entrada

Pontos de entrada têm os mesmos requisitos de definição que as rotinas de inicialização. No entanto, ao contrário das rotinas de inicialização, os pontos de entrada podem retornar um valor. Os tipos de retorno válidos são listados a seguir. Veja a seguir um exemplo de um ponto de entrada:

```
extern "C" __int64 SERVICE_API PrnLog(unsigned long len, const char *val)
{
}
```

Estrutura SERVICE - externa

Para cada serviço de sistema definido, um protótipo de função ECL correspondente deve ser declarado (consulte **Estrutura SERVICE**).

```
servicename := SERVICE
functionname(parameter list) [: keyword = value];
END;
```

```
For example:
email := SERVICE
  simpleSend(String address, String template, String subject)
  : LIBRARY='ec12cw', INITFUNCTION='initEc12Cw';
END;
```

Palavras-Chave

Esta é a lista de palavras-chave válidas para uso em protótipos de funções de serviços:

<i>LIBRARY</i>	Indica o nome do módulo .SO onde o ponto de entrada é definido.
<i>ENTRYPOINT</i>	Especifica um nome para o ponto de entrada. Por padrão, o nome do ponto de entrada é o nome da função.
<i>INITFUNCTION</i>	Especifica o nome da rotina de inicialização definido no módulo que contém o ponto de entrada. No momento, a função de inicialização é chamada uma vez.
<i>INCLUDE</i>	Indica que o protótipo da função está no arquivo incluído especificado. Portanto, o CPP gerado deve incluir o arquivo por meio de #include. Se INCLUDE não for especificado, o protótipo C++ é gerado da definição da função ECL .
<i>C</i>	Indica que o protótipo C== gerado está incluído em um "C" externo, em vez de apenas externo.
<i>PURE</i>	Indica que a função retornará o mesmo resultado todas as vezes que for chamada com os mesmos parâmetros e não apresenta efeitos colaterais. Isso permite que o otimizador execute chamadas mais eficientes à função em alguns casos.
<i>ONCE</i>	Indica que a função não tem efeitos colaterais e é avaliada em tempo de execução da query, mesmo se os parâmetros são constantes. Isso permite que o otimizador execute chamadas mais eficientes à função em alguns casos.
<i>FOLD</i>	Especifica que a função será avaliada em tempo de compilação se todos os parâmetros forem constantes. A especificação de FOLD no SERVICE é aplicada a todas as definições de função no serviço. Nesse caso, NOFOLD pode ser útil para substituir esse padrão para funções individuais que não são adequadas para constant folding (resolução das expressões que podem ser calculadas durante a compilação).
<i>NOFOLD</i>	Especifica que o serviço não é adequado para constant folding.
<i>ACTION</i>	Indica que a função tem efeitos colaterais e exige que o otimizador não remova chamadas à função.
<i>CONTEXT</i>	Somente para uso interno. Indica que um parâmetro de contexto interno extra (ICode-Context *) é passado à função. Esse deve ser o primeiro parâmetro da função.
<i>GLOBALCONTEXT</i>	Somente para uso interno. O mesmo que CONTEXT, mas há restrições sobre onde a função pode ser usada (por exemplo, não pode ser usada em TRANSFORM).
<i>CTXMETHOD</i>	Somente para uso interno. Indica que a função é na verdade um método do contexto de código interno.

Tipo de Dados

Consulte a documentação de mapeamento de ECL para C++ para obter informações sobre o mapeamento de tipos de dados.

Transferindo os Parâmetros Set para um Serviço

Três tipos de parâmetros de conjunto são permitidos: INTEGER, REAL e STRING_n.

INTEGER

Se você quiser consolidar os elementos em um conjunto de inteiros com uma função externa, para declarar a função na estrutura SERVICE:

```
SetFuncLib := SERVICE
  INTEGER SumInt(SET OF INTEGER ss) :
    holertl,library='dab',entrypoint='rtlSumInt';
END;
x:= 3+4.5;
SetFuncLib.SumInt([x, 11.79]); //passed two REAL numbers - it works
```

Para definir a função externa, no arquivo de cabeçalho (.h):

```
__int64 rtlSumInt(unsigned len, __int64 * a);
```

No arquivo de código fonte (.cpp):

```
__int64 rtlSumInt(unsigned len, __int64 * a) {
  __int64 sum = 0;
  for(unsigned i = 0; i < len; i++) {
    sum += a[i];
  }
  return sum;
}
```

O primeiro parâmetro contém o comprimento do conjunto e o segundo é uma matriz de int que mantêm os elementos do conjunto. **Observação:** Na declaração da função na ECL, você também pode ter conjuntos de INTEGER4, INTEGER2 e INTEGER1, mas precisa também alterar o tipo do parâmetro da função C. A relação é:

```
INTEGER8 -- __int64
INTEGER4 -- int
INTEGER2 -- short
INTEGER1 -- char
```

REAL

Se você quer consolidar todos os elementos em um conjunto de números reais:

Para declarar a função na estrutura SERVICE :

```
SetFuncLib := SERVICE
  REAL8 SumReal(SET OF REAL8 ss) :
    holertl,library='dab',entrypoint='rtlSumReal';
END;

INTEGER r1 := 10;
r2 := 20.345;
SetFuncLib.SumReal([r1, r2]);
// intentionally passed an integer to the real set, it works too.
```

Para definir a função externa, no arquivo de cabeçalho (.h):

```
double rtlSumReal(unsigned len, double * a);
```

No arquivo de código fonte (.cpp):

```
double rtlSumReal(unsigned len, double * a) {
  double sum = 0;
  for(unsigned i = 0; i < len; i++) {
    sum += a[i];
  }
  return sum;
}
```


O primeiro parâmetro contém o comprimento do conjunto e o segundo é uma matriz que mantém os elementos do conjunto.

Observação: Você também pode declara a função na ECL como um conjunto de REAL4, mas precisa alterar o parâmetro da função C para float.

STRINGn

Se você quiser calcular a soma dos comprimentos de todas as strings de um conjunto, com os brancos à direita removidos:

Para declarar a função na estrutura SERVICE :

```
SetFuncLib := SERVICE
  INTEGER SumCharLen(SET OF STRING20 ss) :
    holertl,library='dab',entrypoint='rtlSumCharLen';
END;
str1 := '1234567890'+ 'xxxx ' ;
str2 := 'abc';
SetFuncLib.SumCharLen([str1, str2]);
```

Para definir a função externa, no arquivo de cabeçalho (.h):

```
__int64 rtlSumCharLen(unsigned len, char a[ ][20]);
```

No arquivo de código fonte (.cpp):

```
__int64 rtlSumCharLen(unsigned len, char a[ ][20]) {
  __int64 sumtrimmedlen = 0;
  for(unsigned i = 0; i < len; i++) {
    for(int j = 20-1; j >= 0; j--) {
      if(a[i][j] != ' ') {
        break;
      }
      a[i][j] = 0;
    }
    sumtrimmedlen += j + 1;
  }
  return sumtrimmedlen;
}
```

Observação: Na declaração da função C, temos dois parâmetros para o conjunto. O primeiro parâmetro é o comprimento do conjunto, o segundo parâmetro é char[][n], onde n é o SAME do que está em stringn. Por exemplo, se o serviço for declarado como "integer SumCharLen(set of string20)", na função C, o tipo de parâmetro deverá ser char a[][20].

Requerimentos do Plugin

No Windows, os plugins exigem uma função exportada com a seguinte assinatura:

```
Extern "C" _declspec(dllexport) bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb)
```

A função deve preencher a estrutura passada com informações corretas para os recursos do plugin. A estrutura é definida da seguinte forma:

Aviso: Essa função pode ser chamada antes que o plugin seja carregado completamente. Ela não deve fazer nenhuma chamada à biblioteca nem supor que módulos dependentes tenham sido carregados ou que tenha sido inicializada. Mais especificamente: "O sistema não chama DllMain para inicialização e encerramento de processos e linhas de execução. Além disso, o sistema não carrega módulos executáveis adicionais aos quais o módulo especificado faz referência."

```
Struct ECLPluginDefinitionBlock
```

```
{
    Size_t size;
    //size of passed structure - filled in by the calling function
    Unsigned magicVersion ;
    // Filled in by .SO - must be PLUGIN_VERSION (1)
    Const char *moduleName;
    // Name of the module
    Const char *ECL;
    // ECL Service definition for non-HOLE applications
    Unsigned flags;
    // Type of plugin - for user plugin use 1
    Const char *version ;
    // Text describing version of plugin - used in debugging
    Const char *description;
    // Text describing plugin
}
```

Para inicializar as informações em um plugin, use uma variável ou classe global. Ela será construída/destruída adequadamente quando o plugin for carregado/descarregado.

Implantação

.SOs externos devem ser implantados no diretório /opt/HPCCSystems/plugins em cada nó do ambiente de destino. Se forem necessários arquivos de dados externos, eles deverão ser implantados manualmente em cada nó ou um nó de rede deverá fazer referência a eles (o que exige incluir o endereço do .SO) no código. Arquivos implantados manualmente não são incluídos no backup pelos utilitários padrão de backup do SDS.

Restrições

A Refinaria de dados e os Motores de entrega de dados (Thor/Roxie/Doxie) oferecem suporte ao conjunto completo de tipos de dados.

Um exemplo de serviço

O código a seguir mostra um serviço de sistema (.SO) da ECL, denominado examplelib, que contém um ponto de entrada (**stringfind**). Esta é uma versão ligeiramente modificada da função Find encontrada na biblioteca padrão Str. Esta versão foi projetada para funcionar no supercomputador da Refinaria de dados.

Definições ECL

```
EXPORT ExampleLib := SERVICE
    UNSIGNED4 StringFind(CONST STRING src,
        CONST STRING tofind,
        UNSIGNED4 instance )
    : c, pure,entrypoint='elStringFind';
END;
```

Código Módulo .SO:

```
//*****
// hqlplugins.hpp : Defines standard values included
// in
// the plugin header file.
//*****
#ifndef __HQLPLUGIN_INCL
#define __HQLPLUGIN_INCL

#define PLUGIN_VERSION 1
```

```
#define PLUGIN_IMPLICIT_MODULE 1
#define PLUGIN_MODEL_MODULE 2
#define PLUGIN_.SO_MODULE 4

struct ECLPluginDefinitionBlock
{
    size_t size;
    unsigned magicVersion;
    const char *moduleName;
    const char *ECL;
    const char *Hole;
    unsigned flags;
    const char *version;
    const char *description;
};

typedef bool (*EclPluginDefinition) (ECLPluginDefinitionBlock *);

#endif //__HQLPLUGIN_INCL

//*****
// examplelib.hpp : Defines standard values included in
// the plugin code file.
//*****
#ifndef EXAMPLELIB_INCL
#define EXAMPLELIB_INCL

#ifdef _WIN32
#define EXAMPLELIB_CALL __cdecl
#ifdef EXAMPLELIB_EXPORTS
#define EXAMPLELIB_API __declspec(dllexport)
#else
#define EXAMPLELIB_API __declspec(dllimport)
#endif
#else
#define EXAMPLELIB_CALL
#define EXAMPLELIB_API
#endif

#include "hqlplugins.hpp"

extern "C" {
EXAMPLELIB_API bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb);
EXAMPLELIB_API void setPluginContext(IPluginContext * _ctx);
EXAMPLELIB_API unsigned EXAMPLELIB_CALL elStringFind(unsigned srcLen,
    const char * src, unsigned hitLen, const char * hit,
    unsigned instance);
}

#endif //EXAMPLELIB_INCL

//*****
// examplelib.cpp : Defines the plugin code.
//*****
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "examplelib.hpp"

#define EXAMPLELIB_VERSION "EXAMPLELIB 1.0.00"
```

```
static const char * HoleDefinition = NULL;

static const char * EclDefinition =
"export ExampleLib := SERVICE\n"
"  string EchoString(const string src) : c, pure,fold,entrypoint='elEchoString'; \n"
"END;";

EXAMPLELIB_API bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb)
{
    // Warning:      This function may be called without the plugin being loaded fully.
    //              It should not make any library calls or assume that dependent modules
    //              have been loaded or that it has been initialised.
    //
    //              Specifically: "The system does not call DllMain for process and thread
    //              initialization and termination. Also, the system does not load
    //              additional executable modules that are referenced by the specified module."

    if (pb->size != sizeof(ECLPluginDefinitionBlock))
        return false;

    pb->magicVersion = PLUGIN_VERSION;
    pb->version = EXAMPLELIB_VERSION " $Revision: 62376 $";
    pb->moduleName = "lib_examplelib";
    pb->ECL = EclDefinition;
    pb->Hole = HoleDefinition;
    pb->flags = PLUGIN_IMPLICIT_MODULE;
    pb->description = "ExampleLib example services library";
    return true;
}

namespace nsExamplelib {
    IPluginContext * parentCtx = NULL;
}
using namespace nsExamplelib;

EXAMPLELIB_API void setPluginContext(IPluginContext * _ctx) { parentCtx = _ctx; }

//-----

EXAMPLELIB_API unsigned EXAMPLELIB_CALL elStringFind(unsigned srcLen,
const char * src, unsigned hitLen, const char * hit,
unsigned instance)
{
    tgt = (char *)CTXMALLOC(parentCtx, srcLen);
    memcpy(tgt,src,srcLen);
    tgtLen = srcLen;
}
```

Appendix A. Licença Creative Commons

ESTE TRABALHO É FORNECIDO DE ACORDO COM OS TERMOS DA LICENÇA PÚBLICA CREATIVE COMMONS DESCRITOS EM O TRABALHO É PROTEGIDO POR DIREITOS DE AUTOR E / OU OUTRA LEI APLICÁVEL. QUALQUER USO DO TRABALHO QUE NÃO O AUTORIZADO SOB ESTA LICENÇA OU DIREITO AUTURAL É PROIBIDO.

AO EXERCER QUAISQUER DIREITOS AO TRABALHO FORNECIDO AQUI, VOCÊ ACEITA E CONCORDA EM CUMPRIR OS TERMOS DESTA LICENÇA. NA MEDIDA EM QUE ESTA LICENÇA PODE SER CONSIDERADA COMO UM CONTRATO, O LICENCIANTE CONCEDE-NOS OS DIREITOS AQUI CONTIDOS EM CONSIDERAÇÃO À SUA ACEITAÇÃO DE TAIS TERMOS E CONDIÇÕES.

1. Definições

- a. **"Adaptação"** significa um trabalho baseado na Obra, ou na Obra e em outras obras pré-existentes, tais como tradução, adaptação, trabalho derivado, arranjo de música ou outras alterações de um trabalho literário ou artístico, ou fonograma ou performance e inclui adaptações cinematográficas ou qualquer outra forma na qual a Obra possa ser reformulada, transformada ou adaptada, incluindo de qualquer forma reconhecidamente derivada do original, exceto que uma obra que constitua uma Coleção não será considerada uma Adaptação para o propósito desta Licença. Para evitar dúvidas, onde a Obra é uma obra musical, performance ou fonograma, a sincronização da Obra em relação temporizada com uma imagem em movimento ("sincronia") será considerada uma Adaptação para o propósito desta Licença.
- b. **Coleções** significa uma coleção de obras literárias ou artísticas, tais como enciclopédias e antologias, ou performances, fonogramas ou transmissões, ou outras obras ou assuntos que não sejam obras listadas na Seção 1 (f) abaixo, que, em razão da seleção e arranjo de seus conteúdos, constituem criações intelectuais, nas quais a Obra é incluída em sua totalidade de forma não modificada junto com uma ou mais contribuições, cada uma constituindo em si mesmas obras independentes e independentes, que juntas são reunidas em um todo coletivo. Um trabalho que constitua uma Coleção não será considerado uma Adaptação (conforme definido acima) para os propósitos desta Licença.
- c. **DISTRIBUTE** meios para disponibilizar ao público o original e cópias da Obra através da venda ou outra transferência de propriedade.
- d. **"Licenciante"** significa o indivíduo, indivíduos, entidades ou entidades que oferecem o (s) Trabalho sob os termos desta Licença.
- e. **"Autor original"** significa, no caso de uma obra literária ou artística, o indivíduo, indivíduos, entidade ou entidades que criaram a Obra ou, se nenhum indivíduo ou entidade puder ser identificado, o editor; e além disso (i) no caso de uma performance, os atores, cantores, músicos, dançarinos e outras pessoas que atuam, cantam, transmitem, declamam, tocam, interpretam ou executam obras literárias ou artísticas ou expressões de folclore; (ii) no caso de um fonograma, o produtor sendo a pessoa ou entidade legal que primeiro fixa os sons de uma performance ou outros sons; e, (iii) no caso de transmissões, a organização que transmite a transmissão.
- f. **"Trabalhos"** significa o trabalho literário e / ou artístico oferecido sob os termos desta Licença, incluindo, sem limitação, qualquer produção no domínio literário, científico e artístico, qualquer que seja o modo ou forma de sua expressão, incluindo a forma digital, como um livro, panfleto e outra escrita; uma palestra, discurso, sermão ou outro trabalho da mesma natureza; um trabalho dramático ou dramático-musical; um trabalho coreográfico ou entretenimento em show mudo; uma composição musical com ou sem palavras; uma obra cinematográfica à qual são assimilados trabalhos expressos por um processo análogo à cinematografia; uma obra de desenho, pintura, arquitetura, escultura, gravura ou litografia; um trabalho fotográfico ao qual são assimilados trabalhos expressos

por um processo análogo à fotografia; um trabalho de arte aplicada; uma ilustração, mapa, plano, esboço ou trabalho tridimensional relativo à geografia, topografia, arquitetura ou ciência; uma performance; uma transmissão; um fonograma; uma compilação de dados na medida em que é protegido como um trabalho com direitos autorais; ou um trabalho realizado por uma variedade ou artista de circo, na medida em que não é considerado um trabalho literário ou artístico.

- g. **VOCÊ** significa uma pessoa física ou jurídica que exerce direitos sob esta Licença e que não tenha anteriormente violado os termos desta Licença em relação à Obra, ou que tenha recebido permissão expressa da Licenciante para exercer direitos sob esta Licença, apesar de uma violação anterior.
- h. **"Execução pública"** meios para realizar recitações públicas da Obra e para comunicar ao público as recitações públicas, por qualquer meio ou processo, incluindo por fio ou meio sem fio ou apresentações digitais públicas; disponibilizar ao público Obras de tal maneira que os membros do público possam acessar estas Obras de um lugar e em um lugar escolhido individualmente por eles; realizar a Obra ao público por qualquer meio ou processo e a comunicação ao público das performances da Obra, inclusive por meio de performance digital pública; para transmitir e retransmitir o trabalho por qualquer meio, incluindo sinais, sons ou imagens.
- i. **"Reproduzir"** meios para fazer cópias da Obra por qualquer meio, incluindo sem limitação por gravações sonoras ou visuais e o direito de fixação e reprodução de fixações da Obra, incluindo armazenamento de uma performance protegida ou fonograma em formato digital ou outro meio eletrônico.

2. Direitos de Negociação. Nada nesta Licença destina-se a reduzir, limitar ou restringir quaisquer usos livres de direitos autorais ou direitos decorrentes de limitações ou exceções previstas em conexão com a proteção de direitos autorais sob a lei de direitos autorais ou outras leis aplicáveis.

3. Concessão de licença. Sujeito aos termos e condições desta Licença, a Licenciante concede a você uma licença mundial, livre de royalties, não exclusiva, perpétua (para a duração dos direitos autorais aplicáveis) para exercer os direitos na Obra, conforme estabelecido abaixo:

- a. reproduzir o trabalho, incorporar o trabalho em uma ou mais coleções e reproduzir o trabalho incorporado às coleções; e,
- b. para distribuir e executar publicamente o trabalho, inclusive como incorporado em coleções.
- c. Para evitar dúvidas:
 - 1. **Esquemas obrigatórios de licença não-renunciáveis.** Nas jurisdições em que o direito de cobrar royalties por meio de qualquer esquema de licenciamento estatutário ou compulsório não puder ser renunciado, o Licenciante reserva-se o direito exclusivo de cobrar tais royalties para qualquer exercício por Você dos direitos concedidos sob esta Licença;
 - 2. **Esquemas obrigatórios de licença não-renunciáveis.** Nas jurisdições em que o direito de cobrar royalties por meio de qualquer esquema de licenciamento estatutário ou compulsório não puder ser renunciado, o Licenciante reserva-se o direito exclusivo de cobrar tais royalties para qualquer exercício por Você dos direitos concedidos sob esta Licença;
 - 3. **Esquemas de Licença Voluntária.** Nas jurisdições em que o direito de cobrar royalties por meio de qualquer esquema de licenciamento estatutário ou compulsório não puder ser renunciado, o Licenciante reserva-se o direito exclusivo de cobrar tais royalties para qualquer exercício por Você dos direitos concedidos sob esta Licença;

Os direitos acima podem ser exercidos em todas as mídias e formatos, agora conhecidos ou futuramente. Os direitos acima incluem o direito de fazer as modificações que forem tecnicamente necessárias para exercer os direitos em outras mídias e formatos, mas caso contrário você não terá direitos para fazer Adaptações. Sujeito à Seção 8 (f), todos os direitos não expressamente concedidos pelo Licenciante ficam aqui reservados.

4. Restrições A licença concedida na Seção 3 acima está expressamente sujeita e limitada pelas seguintes restrições:

- a. Você pode distribuir ou executar publicamente o trabalho somente sob os termos desta licença. Você deve incluir uma cópia, ou o Identificador Uniforme de Recursos (URI) para esta Licença, com cada cópia do Trabalho que Você Distribuir ou Realizar publicamente. Você não pode oferecer ou impor quaisquer termos na Obra que restrinjam os termos desta Licença ou a capacidade do destinatário da Obra de exercer os direitos concedidos a esse destinatário sob os termos da Licença. Você não pode sublicenciar o Trabalho. Você deve incluir uma cópia, ou o Identificador Uniforme de Recursos (URI) para esta Licença, com cada cópia do Trabalho que Você Distribuir ou Realizar publicamente. Você não pode oferecer ou impor quaisquer termos na Obra que restrinjam os termos desta Licença ou a capacidade do destinatário da Obra de exercer os direitos concedidos a esse destinatário sob os termos da Licença. Esta Seção 4 (a) aplica-se à Obra como incorporada em uma Coleção, mas isso não exige que a Coleção, além da Obra, fique sujeita aos termos desta Licença. Se você criar uma coleção, mediante notificação de qualquer licenciante, deverá, na medida do possível, retirar da coleção qualquer crédito, conforme exigido pela Seção 4 (b), conforme solicitado.
- b. Se Você Distribuir, ou Executar Publicamente o Trabalho ou Coleções, Você deve, a menos que uma solicitação tenha sido feita conforme a Seção 4 (a), manter intactos todos os avisos de direitos autorais para o Trabalho e fornecer, razoável para o meio ou meio que Você está utilizando: (i) o nome do Autor Original (ou pseudônimo, se aplicável) se fornecido, e / ou se o Autor Original e / ou o Licenciante designar outra parte ou partes (por exemplo, um instituto patrocinador, entidade editorial, revista) para atribuição ("Partes de Atribuição") no aviso de direitos autorais do Licenciante, nos termos de serviço ou por outros meios razoáveis, o nome de tal parte ou partes; (ii) o título do Trabalho, se fornecido; (iii) na medida do razoavelmente praticável, o URI, se houver, que o Licenciante especificar como associado ao Trabalho, a menos que tal URI não se refira ao aviso de direitos autorais ou informações de licenciamento para o Trabalho. O crédito exigido por esta Seção 4 (b) pode ser implementado de qualquer maneira razoável; desde que, no caso de uma Cobrança, no mínimo tal crédito apareça, se um crédito para todos os autores contribuintes da Coleção aparecer, então como parte desses créditos e de uma forma pelo menos tão proeminente quanto os créditos para os outros autores contribuintes. Para evitar dúvidas, Você só pode usar o crédito exigido por esta Seção para fins de atribuição da maneira descrita acima e, ao exercer Seus direitos sob esta Licença, Você não pode implícita ou explicitamente afirmar ou sugerir qualquer conexão com, patrocínio ou endosso pelo Autor Original, Licenciante e / ou Partes da Afiliação, conforme o caso, de Você ou Seu uso do Trabalho, sem a prévia autorização expressa por escrito do Autor Original, Licenciante e / ou Partes de Atribuição.
- c. Salvo acordo em contrário por escrito pelo Licenciador ou conforme seja permitido pela lei aplicável, se Você Reproduzir, Distribuir ou Executar Publicamente o Trabalho por si próprio ou como parte de quaisquer Coleções, Você não deve distorcer, mutilar, modificar ou tomar outro derogatória em relação ao Trabalho que seria prejudicial à honra ou reputação do Autor Original.

5. Representações, Garantias e Isenção de Responsabilidade

A MENOS QUE, DE ACORDO COM AS PARTES, O LICENCIANTE DECLARA AS TRABALHOS COMO ESTÁ E NÃO FAZ REPRESENTAÇÃO OU GARANTIA DE QUALQUER TIPO RELATIVO AO TRABALHO, EXPRESSA, IMPLÍCITA, ESTATUTÁRIA OU OUTRA, INCLUINDO, SEM LIMITAÇÃO, GARANTIAS DE TÍTULO, COMERCIALIZABILIDADE, APTIDÃO PARA UMA FINALIDADE ESPECÍFICA, NÃO-VIOLAÇÃO OU AUSÊNCIA DE DEFEITOS OU PREJUÍZOS LATENTES OU OUTROS, EXATIDÃO OU PRESENÇA DE AUSÊNCIA DE ERROS, SEJAM DESCOBERTOS. ALGUMAS JURISDIÇÕES NÃO PERMITEM A EXCLUSÃO DE GARANTIAS IMPLÍCITAS, TAL EXCLUSÃO PODE NÃO SE APLICAR A VOCÊ.

6. Limitação de Responsabilidade. EXCETO NA MEDIDA EXIGIDA PELA LEGISLAÇÃO APLICÁVEL, EM NENHUM CASO, A LICENCIANTE SERÁ RESPONSÁVEL POR QUALQUER TEORIA LEGAL POR QUALQUER DANOS ESPECIAIS, INCIDENTAIS, CONSEQUENTES, PUNITIVOS OU EXEMPLARES DECORRENTES DESTA LICENÇA OU O USO DA OBRA, MESMO QUE O LICENCIADOR TENHA SIDO AVISADO DA POSSIBILIDADE DE TAIS DANOS.

7. Encerramento

- a. Esta Licença e os direitos aqui concedidos serão automaticamente rescindidos mediante qualquer violação por Você dos termos desta Licença. Indivíduos ou entidades que receberam Coleções sob esta Licença, no entanto, não terão

suas licenças terminadas, desde que tais indivíduos ou entidades permaneçam em total conformidade com essas licenças. As seções 1, 2, 5, 6, 7 e 8 sobreviverão a qualquer término desta Licença.

- b. Sujeito aos termos e condições acima, a licença concedida aqui é perpétua (para a duração dos direitos autorais aplicáveis na Obra). Não obstante o acima, o Licenciador se reserva o direito de liberar o Trabalho sob diferentes termos de licença ou de parar de distribuir o Trabalho a qualquer momento; desde que, no entanto, tal escolha não sirva para retirar esta Licença (ou qualquer outra licença que tenha sido ou deva ser concedida sob os termos desta Licença), e esta Licença continuará em pleno vigor e efeito, a menos que seja rescindida como indicado acima.

8. Miscelânea

- a. Cada vez que você distribuir ou executar publicamente o trabalho ou uma coleção, o licenciante oferece ao destinatário uma licença para o trabalho nos mesmos termos e condições que a licença concedida a você sob esta licença.
- b. Se qualquer disposição desta Licença for inválida ou inexecutável sob a lei aplicável, ela não afetará a validade ou aplicabilidade do restante dos termos desta Licença, e sem qualquer ação adicional por parte das partes deste contrato, tal disposição será reformada para a extensão mínima necessária para tornar tal provisão válida e executável.
- c. Nenhum termo ou disposição desta Licença será considerado renunciado e nenhuma violação será consentida, a menos que tal renúncia ou consentimento seja feito por escrito e assinado pela parte a ser acusada de tal renúncia ou consentimento.
- d. Esta Licença constitui o acordo integral entre as partes com relação ao Trabalho licenciado aqui. Não há entendimentos, acordos ou representações com relação ao Trabalho não especificado aqui. O Licenciante não será obrigado por quaisquer disposições adicionais que possam aparecer em qualquer comunicação sua. Esta Licença não pode ser modificada sem o acordo mútuo por escrito do Licenciador e Você.
- e. Os direitos concedidos sob, e a matéria mencionada, nesta Licença foram redigidos utilizando a terminologia da Convenção de Berna para a Proteção de Obras Literárias e Artísticas (como emendada em 28 de setembro de 1979), a Convenção de Roma de 1961, a WIPO Copyright Tratado de 1996, o Tratado da OMPI sobre Prestações e Fonogramas de 1996 e a Convenção Universal dos Direitos de Autor (revista em 24 de Julho de 1971). Esses direitos e objeto entram em vigor na jurisdição relevante em que os termos da Licença são solicitados, de acordo com as disposições correspondentes da implementação dessas disposições do tratado na legislação nacional aplicável. Se o conjunto padrão de direitos concedidos sob a lei de direitos autorais aplicável incluir direitos adicionais não concedidos sob esta Licença, esses direitos adicionais serão considerados incluídos na Licença; esta Licença não tem a intenção de restringir a licença de quaisquer direitos sob a lei aplicável.

Index

Symbols

#APPEND, 391
#BREAK, 406
#CONSTANT, 392
#DECLARE, 393
#DEMANGLE, 394
#ELSE, 404
#ELSEIF, 404
#END, 406
#ERROR, 395
#EXPAND, 396
#EXPORT, 397
#EXPORTXML, 400
#FOR, 402
#GETDATATYPE, 403
#IF, 404
#INMODULE, 405
#LOOP, 406
#MANGLE, 407
#ONWARNING, 408
#OPTION, 409
#SET, 419
#STORED, 420
#TEXT, 421
#UNIQUENAME, 422
#WARNING, 424
#WEBSERVICE, 425
#WORKUNIT, 164, 426
, e, 340
, insertion sort, 340
.ECL files, 25
.SO, 430
__COMPRESSED__, 69
__CONTAINERIZED__, 13
__ECL_LEGACY_MODE__, 13
__ECL_VERSION_MAJOR__, 13
__ECL_VERSION_MINOR__, 13
__ECL_VERSION__, 13
__OS__, 13
__PLATFORM__, 13
__STAND_ALONE__, 13
__TARGET_PLATFORM__, 13

, 158, 184

A

A estrutura TRANSFORM, 117
ABS, 145
ABS function, 145
Ações como Definições, 29

ACOS, 146
ACOS function, 146
action, 338
Addition, 30
afinidade de processador, 412
afinidade de processador do Roxie, 412
AFTER, 151
AGGREGATE, 147
AGGREGATE function, 147
alinhar um conjunto de dados, 74
ALL, 104, 185, 220, 238, 278, 286, 290, 373
ALL keyword, 104
ALLNODES, 150
ALLNODES function, 150
AND, 38, 59
AND NOT, 59
anexação de CPU, 412
anexação de CPU do Roxie, 412
ANY, 22
APPLY, 151
APPLY function, 151
argumento, 20
arithmetic operators, 30
AS, 108
As Expressões, 30
ASCII, 70, 152, 282
ASCII function, 152
ASIN, 153
ASIN function, 153
ASSERT, 154
ASSERT function, 154
ASSTRING, 156
ASSTRING function, 156
ATAN, 157
ATAN function, 157, 158
ATAN2, 158
ATMOST, 238, 290
AVE, 159
AVE function, 159

B

BEFORE, 151
BEGINC++, 120
BEST, 185, 290, 355
BETWEEN, 38
Between Operator, 38
BIG_, 40
Binary values, 12
Bitshift Left, 31
Bitshift operators, 30
Bitshift Right, 31
Bitwise AND, 30
Bitwise Exclusive OR, 30

Bitwise NOT, 30
Bitwise operators, 30
Bitwise OR, 30
BLOB in INDEX, 60
Boolean, 16
BOOLEAN, 39, 93
Boolean AND, 32
Boolean Definition, 16
Boolean NOT, 32, 32
Boolean OR, 32
BOOLEAN value type, 39
BUILD, 160
BUILD action, 85, 87

C

Cada definição ECL, 14
CASE, 166, 290
CASE function, 166
casting operator, 54
Casting Rules, 55
CATCH, 167
CATCH Function, 167
Character Sets, 17
CHECKPOINT, 376
Child Dataset, 77
CHOOSE, 169
CHOOSE function, 169
CHOOSEN, 67, 77, 170
CHOOSEN function, 170
CHOOSESETS, 171
CHOOSESETS function, 171
CLUSTER, 161, 280, 281, 283, 284, 382
CLUSTERSIZE, 172
colchetes, 9, 20
COMBINE, 173
COMBINE function, 173
compactado, 42
comparison operator, 31
COMPRESSED, 69, 85, 280
COMPRESSION, 161
Concatenação, 36
concatenação, 78
CONST, 20, 154, 429
CONST Function, 429
Constant set, 16
constant values, 16
constantes, 11
constantes de string, 11
constants, 13
consulta externa, 160
consulte as, 27
CORRELATION, 176
CORRELATION function, 176

COS, 178
COS function, 178
COSH, 179
COSH function, 179
COUNT, 67, 77, 180, 253, 253, 327, 389
COUNT function, 180
COUNTER, 78, 142, 190, 218, 243, 261, 272, 303
COVARIANCE, 182
COVARIANCE function, 182
CRON, 184
CRON function, 184
CSV, 70, 73, 278, 281, 285, 295
CSV Files, 70, 281

D

DATA, 47
Data string, 11
DATA value type, 47
Dataset, 17
DATASET, 67, 67, 70, 79, 89, 161, 335
DATASET declaration, 88, 89
DATASET filho, 77
DATASET parameter, 21
DATASET parameters, 69
Datasets filhos aninhados, 91
DECIMAL, 42
DECIMAL value type, 42
DEDUP, 161, 185, 264, 266
DEDUP function, 185, 187, 317
DEFAULT, 60
DEFINE, 188
DEFINE function, 188
Definição BOOLEAN, 19
Definições como ações, 29
Definition Name, 14
Definition Operator, 14
Definition Types, 16
Definition Visibility, 25, 137
DENORMALIZE, 189
DENORMALIZE function, 189
DEPRECATED, 377
DEPRECATED workflow service, 377
DESCEND, 308, 309
DICTIONARY, 83
DICTIONARY parameter, 21
DISTINCT no SQL, 186
DISTRIBUTE, 161, 192
DISTRIBUTE function, 192
DISTRIBUTED, 85, 161, 195
DISTRIBUTED function, 195
DISTRIBUTION, 196
DISTRIBUTION action, 371
DISTRIBUTION function, 196

Divisão por zero, 30
Division, 30
dot syntax, 27
Dynamic Files, 89

E

EBCDIC, 70, 198, 282
EBCDIC function, 198
ECL, 10
ECL IDE, 11
ECL keywords, 14
EMBED, 125
EMBED Structure, 125
ENCODING, 337
ENCRYPT, 69, 70, 71, 72, 280, 281, 283, 284
ENDC++, 120
ENDEMBED, 125
ENDMACRO, 135
ENTH, 171, 199
ENTH function, 199
ENUM, 53
ENUM datatype, 53
Equivalence, 31, 98
Equivalence Comparison, 31
ERROR, 200
ERROR function, 200
ESCAPE, 70
Estrutura BEGINC++, 120
Estrutura MACRO, 135
Estrutura RECORD, 33
estrutura RECORD, 102
Estrutura SERVICE, 430
estrutura TRANSFORM, 210
estrutura TYPE, 57
estruturas RECORD aninhadas, 75
EVALUATE, 201
EVALUATE action, 201
EVALUATE function, 202
EVENT, 203
EVENT function, 203
EVENTEXTRA, 205
EVENTEXTRA function, 205
EVENTNAME, 204
EXCEPT, 105
EXCEPT keyword, 105
EXCLUSIVE, 171
EXISTS, 206
EXISTS function, 206
EXP, 207
EXP function, 207
EXP Function, 255
EXPIRE, 161, 246, 247, 280, 281, 283, 284, 382
Explicit Casting, 54

EXPORT, 25, 93, 106
EXPORTed, 27
Expression, 14
Expressões, 76
Expressões como Ações, 29
Expressões e Operadores, 30
EXTEND, 278, 286
Extended PARSE, 292
Extended PARSE Examples, 292
External Service, 430
external system services, 430

F

FAIL, 154, 167, 208
FAIL action, 200, 208
FAILCODE, 209
FAILCODE function, 209
FAILMESSAGE, 167, 210, 228, 336
FAILMESSAGE function, 210
FAILURE, 378
FAILURE workflow, 209
FAILURE workflow service, 378
FALSE, 39, 118, 118
FETCH, 211
FETCH function, 211
FEW, 147, 161, 170, 217, 238, 339, 349, 379, 386
field sequence, 425
File Scope, 88
FILEPOSITION, 161
Filtros, 19
FIRST, 85, 161, 290
FLAT, 69
Flat Files, 280
FOLD, 431
Foreign files, 88
FORMAT, 282
FORWARD, 137
forward reference, 10
FROM, 108
FROMJSON, 213
FROMJSON function, 213
FROMUNICODE, 214
FROMUNICODE function, 214
FROMXML, 215
FROMXML function, 215
FULL ONLY, 242
FULL OUTER, 242
função, 208
Função, 337
Função TRANSFORM, 302
FUNCTION, 127
function LIBRARY, 164
FUNCTION Structure, 127

FUNCTIONMACRO, 130
FUNCTIONMACRO Structure, 130
Functions, 20

G

GETENV, 216
GETENV function, 216
GETISVALID, 93
GLOBAL, 217, 379
GLOBAL function, 217
GLOBAL workflow service, 217
Gráficos Lógicos, 412
GRAPH, 218
GRAPH function, 218
Greater or Equal, 31
Greater Than, 31
GROUP, 107, 173, 189, 295, 317
Group, 220
GROUP function, 220, 363
GROUP keyword, 107, 366
GROUPED, 67, 79, 238

H

HASH, 185, 222, 238
HASH function, 222
HASH32 function, 223
HASH64, 223, 224
HASH64 function, 224
HASHCRC, 225
HASHCRC function, 225
HASHMD5, 226
HASHMD5 function, 226
HAVING, 227
HAVING function, 227
HEADING, 70, 282, 283, 285, 336
heapsort, 340
Hexadecimal, 11
hexadecimal, 12
hexadecimal compactado, 47
HPCC, 10
hthor, 340
HTTPCALL, 228
HTTPCALL Function, 228
HTTPHEADER, 228, 337

I

IF, 230
IF function, 230
IFBLOCK, 57
IFF, 231
IFF function, 231
Implicit Casting, 54
Implicit Dataset, 91

IMPORT, 108
IMPORT AS, 108
IMPORT FROM, 108
IMPORT function, 232
IMPORTed, 27
IN, 37
In Line Dataset, 74
In-Line Dataset, 74
INCLUDE, 431
INDEPENDENT, 380
INDEPENDENT workflow service, 380
INDEX, 85
INDEX declaration, 85
Indexing, 17
Inline TRANSFORMs, 142
INNER, 267
INTEGER, 12, 40, 93, 432
Integer Division, 30
INTEGER value type, 40
INTERFACE, 132
interface, 347
INTERFACE Structure, 132
INTERNAL, 251
INTFORMAT, 233
INTFORMAT function, 233
ISVALID, 234
ISVALID function, 234
ITERATE, 235
ITERATE function, 235

J

JOIN, 237, 243, 243
JOIN function, 237
JOIN Set, 243
JOIN setofdasetts, 244
Join Types, 267
joincondition, 238
JOINED, 339
joinflags, 238
JOINS FULL OUTER, 242
JSON, 72, 278, 284
JSON Files, 284
junção para, 110

K

KEEP, 185, 238, 290
KEYDIFF, 246
KEYDIFF function, 246
KEYED, 110, 159, 176, 180, 182, 206, 238, 253, 253, 268, 301, 348, 349, 366
KEYED e, 110
Keyed JOIN, 241
KEYPATCH, 247

KEYPATCH action, 246
KEYPATCH function, 247
KEYUNICODE, 249
KEYUNICODE function, 249

L

Landing Zone files, 89
LAST, 171
LEFT, 112, 190, 271
LEFT ONLY, 242, 267
LEFT OUTER, 242, 267
LENGTH, 67, 77, 250
LENGTH function, 250
Less or Equal, 31
Less Than, 31
LIBRARY, 137, 251
LIBRARY function, 251
LIKELY, 113
LIMIT, 238, 253
LIMIT function, 253
LINKCOUNTED, 67, 79
LITERAL, 336
LITTLE_ENDIANLITTLE_ENDIAN, 40
LN, 255
LN function, 207, 255
LOAD, 93
LOADXML, 256
LOADXML function, 256
LOCAL, 26, 147, 161, 173, 185, 189, 199, 211, 220, 235, 238, 258, 264, 299, 301, 317, 339, 349, 355
LOCAL function, 85, 258
LOCALE, 57
LOG, 259, 337
LOG function, 259
LOGICAL Filenames, 88
logical operators, 32, 59
LOOKUP, 238
LOOP, 260
LOOP function, 218, 260
loopbody, 260
loopcondition, 260
loopcount, 260
loopfilter, 260
LZW, 85, 161

M

MACRO, 135
MANY, 147, 217, 238, 290, 349
MAP, 262
MAP function, 262
MATCHED, 100, 290
MATCHED ALL, 290
MATCHLENGTH, 100

MATCHPOSITION, 100
MATCHROW, 100
MATCHTEXT, 100
MATCHUNICODE, 100
MAX, 290, 334
MAX function, 263
MAXCOUNT, 60
MAXLENGTH, 57, 60, 70, 85, 85, 85, 93, 161, 289
memory exhausted, 241
MERGE, 161, 264, 336, 349
MERGE function, 264
MERGEJOIN, 266
MERGEJOIN function, 243, 266
MIN, 268, 290
MIN function, 268
MODULE, 137
MODULE Structure, 137
Modulus Division, 30
MOFN, 267
MULTIPLE, 382
Multiplication, 30

N

n, 41
N-ary merge/join, 218
Name, 14
NAMED, 23, 196, 278, 286, 286
NAMED OUTPUT, 286
Named Output Dataset, 74
NAMESPACE, 336
Natural Language Parsing, 95
No entanto, os valores, 117
NOCASE, 290, 311, 312, 313
NOFOLD, 274, 431
NOFOLD function, 274
NOLOCAL, 269
NOLOCAL function, 269
non-procedural language, 10
NONEMPTY, 270
NONEMPTY function, 270
NORMALIZE, 271
NORMALIZE function, 271
NOROOT, 71, 72, 161
NOSCAN, 290
NOSORT, 189, 238
Not Equal, 31
NOT MATCHED, 290
NOT MATCHED ONLY, 290
NOTHOR, 275
NOTHOR action, 275
NOTIFY, 276
NOTIFY function, 276
NOTRIM, 70, 336

NOXPATH, 278

O

O suporte XPATH, 61
ONCE, 431
ONFAIL, 167, 228, 253, 336
ONWARNING, 381
ONWARNING workflow service, 381
Opções HTTPCALL, 228
Opções SOAPCALL, 336
operador de comparação, 364
Operador IN, 37
Operadores Record Set, 33
Operators, 30
OPT, 69, 85, 110, 283, 285, 303
OR, 59
ORDERED, 277
ORDERED action, 277
OUTPUT, 278, 280, 281, 283, 284, 286, 286, 295
OUTPUT - CSV Files, 281
OUTPUT - JSON Files, 284
OUTPUT - NAMED Files, 286
OUTPUT - XML Files, 283
OUTPUT action, 278
OUTPUT Pipe Files, 285
OUTPUT Scalar Values, 286
OUTPUT Thor/Flat Files, 280
OUTPUT Workunit Files, 287
OVERWRITE, 161, 246, 247, 280, 281, 283, 284

P

PACKED, 57
packed decimal, 42, 42
PARALLEL, 288, 301, 336
PARALLEL function, 288
Parameter Passing, 20
parameters, 14
PARSE, 289, 290
PARSE Examples, 292
PARSE function, 100, 289
PARSE Text, 289
PARSE XML, 291
PARTITION LEFT, 238
PARTITION RIGHT, 238
Passing Set Parameters, 431
PATTERN, 96
Perl regular expression, 311, 313
PERSIST, 382
PERSIST workflow service, 382
PHYSICALENGTH, 93
Pipe, 73
PIPE, 278, 285, 295
PIPE Files, 73

PIPE function, 74, 295
Pool memory exhausted, 241
POWER, 297
POWER function, 297
precisa usar no mínimo um ou dois parâmetros, 242
PREFETCH, 301, 301
PRELOAD, 69, 298
PRELOAD function, 298
PRIORITY, 384
PRIORITY workflow service, 384
PROCESS, 299
PROCESS function, 299
PROJECT, 301, 303
PROJECT function, 301
PULL, 305
PULL function, 305
PURE, 431

Q

QSTRING, 44
QSTRING string constants, 11
QSTRING value type, 44
query library, 137
quicksort, 340
QUOTE, 70, 282

R

Racionalidade Implícita do Dataset, 91
RANDOM, 306
RANDOM function, 306
RANGE, 307
RANGE function, 307
RANK, 308
RANK function, 308
RANKED, 309
RANKED function, 309
REAL, 41, 432
REAL data type, 41
REALFORMAT, 310
REALFORMAT Function, 310
realvalue, 322
RECORD, 57, 72, 73
record matching, 266
Record Matching Logic, 266
Record Set, 16, 17, 19, 33
Record Set Definition, 17
RECORD Structure, 57, 115
RECORD structure, 57, 92, 100, 141, 290, 291, 301, 349, 350, 366
RECORDOF, 52
RECORDOF datatype, 52
RECOVERY, 385
RECOVERY workflow service, 385

- recstruct, 323
- referência antecipada, 188
- referência futura, 98
- REFRESH, 382
- regex, 311, 312, 313
- REGEXFIND, 311
- REGEXFIND function, 311
- REGEXFINDSET, 312
- REGEXFINDSET function, 312
- REGEXREPLACE, 313
- REGEXREPLACE function, 313
- REGROUP, 314
- REGROUP function, 314
- regular expression, 96
- REJECTED, 316
- REJECTED function, 316, 370
- Relationality, 91
- REPEAT, 285, 295
- Requerimentos da Função TRANSFORM, 299, 302
- Requisitos TRANSFORM, 302
- Requisitos TRANSFORM PROJECT, 302
- Reserved Words, 14
- resultrec, 323
- RETRY, 228, 336
- RETURN, 28, 127, 130
- RIGHT, 112, 190
- RIGHT ONLY, 242
- RIGHT OUTER, 242
- RIGHT1, 147
- RIGHT2, 147
- ROLLUP, 317, 340
- ROLLUP function, 317
- ROUND, 321
- ROUND function, 321
- ROUNDUP, 322
- ROUNDUP function, 322
- ROW, 85, 139, 161, 323
- ROW function, 323
- ROWDIFF, 327
- ROWDIFF function, 327
- ROWS(LEFT), 114
- ROWS(RIGHT), 114
- ROWSET, 218
- ROWSET(LEFT), 218
- Roxie, 340
- RULE, 96
- S**
- SAMPLE, 328
- SAMPLE function, 328
- Scalar OUTPUT, 286
- SCAN, 290
- SCAN ALL, 290
- Scope, 14
- SCOPE, 88
- SELF, 115, 115, 140
- SEPARATOR, 70, 282
- SEQUENTIAL, 286, 329
- SEQUENTIAL function, 329
- Service Function Keywords, 431
- SERVICE Structure, 151, 427
- SERVICE structure, 432, 433
- Serviço de fluxo de trabalho CHECKPOINT, 376
- Serviço de Fluxo de Trabalho GLOBAL, 379
- Set, 16
- SET, 20, 330
- Set Definition, 16, 16
- SET function, 16, 330
- SET OF, 50
- SET OF value type, 50
- Set Operators, 35
- Set Ordering, 17
- SET parameters, 20
- Set Parameters, 431
- Sets can contain definitions and expressions, 16
- SHARED, 25, 116
- Shared Object, 430
- SIN, 332
- SIN function, 332
- SINGLE, 282, 382
- Single-Row Dataset, 76
- SINH, 333
- SINH function, 333
- SIZEOF, 334
- SIZEOF function, 334
- SKEW, 161, 192, 238, 339, 349
- SKIP, 117, 140, 167, 238, 253, 290
- SMART, 238
- SOAPACTION, 336
- SOAPCALL, 62, 335, 337
- SOAPCALL Function, 335
- SORT, 339
- SORT function, 339
- SORTED, 85, 161, 237, 243, 264, 266, 343
- SORTED function, 343
- SQRT, 344
- SQRT function, 344
- square brackets, 16, 50
- STABLE, 339
- STEPPED, 345
- STEPPED function, 345
- STORE, 93
- STORED, 347, 386
- STORED function, 347
- STORED workflow service, 386
- STREAMED, 67, 79
- String, 17

STRING, 43
string operator, 36
string slice, 17
string UTF8, 11
STRING value type, 43
STRINGn, 433
substring, 17
Subtraction, 30
SUCCESS, 388
SUCCESS workflow service, 388
SUM, 348
SUM function, 348
SuperFile, 89
system constants, 13

T

TABLE, 27, 349
TABLE function, 28
TABLE Function, 349
TAN, 351
TAN Function, 351
TANH, 352
TANH Function, 352
Template Language, 390
Temporary SuperFile, 89
TERMINATOR, 70, 282
THISNODE, 353
THISNODE Function, 353
THOR, 69, 278, 287
Thor, 340
THRESHOLD, 161, 238, 339, 349
TIMELIMIT, 228, 336
TIMEOUT, 228, 336
Tipos de valores, 39
TOJSON, 354
TOJSON function, 354
TOKEN, 96
Tomita parsing, 290
TOPN, 355
TOPN Function, 355
TOUNICODE, 356
TOUNICODE Function, 356
TOXML, 357
TOXML function, 357
TRACE, 358
traceEnabled, 358
traceLimit, 358
TRANSFER, 360
TRANSFER Function, 360
TRANSFORM, 140, 209
transform function, 211, 235
TRANSFORM Function, 299
Transform Requirement Process, 299

Transform Requirements, 299
TRANSFORM structure, 115, 140, 323
TRANSFORMs em linha, 142
tratando DICTIONARY como um DATASET, 81
TRIM, 228, 283, 285, 336, 361
TRIM Function, 361
TRIM OPT, 283, 285
TRUE, 39, 118, 118
TRUNCATE, 362
TRUNCATE Function, 362
TYPE, 92
Type Casting, 54
TYPE structure, 92
Type Transfer, 54
TypeDef, 16
TypeDef Definition, 18
TYPEOF, 51
TYPEOF data type, 51

U

UDECIMALn, 42
Uma expressão Perl regular padrão., 312
UNGROUP, 363
UNGROUP Function, 363
Unicode, 11
UNICODE, 45, 70, 282
UNICODE value type, 45
UNICODEORDER, 364
UNICODEORDER function, 364
UNLIKELY, 113
UNORDERED, 238, 365
UNORDERED function, 365
UNSIGNED, 40, 42
UNSIGNED value type, 40
UNSORTED, 69, 349
UNSTABLE, 339
UPDATE, 161, 280, 281, 283, 284
USE, 290
UTF-8, 11
UTF8, 46
UTF8 value type, 46

V

valores constantes, 75
Valores Decimais, 12
Valores não-definidos, 12
Value, 16
Value Definition, 16
Value Types, 14, 20
VARIANCE, 366
VARSTRING, 48
VARSTRING string constants, 11
VARSTRING value type, 48

VARUNICODE, 49
VARUNICODE value type, 49
Virtual, 60
VIRTUAL, 137
VIRTUAL EXPORT, 106
Virtual fileposition, 60
Virtual localfileposition, 60
Virtual logicalfilename, 60
VIRTUAL SHARED, 116

W

WAIT, 368
WAIT Function, 368
WHEN, 369, 389
WHEN Function, 369
WHEN serviço de fluxo de trabalho, 203
WHEN workflow service, 389
WHICH, 370
WHICH function, 316
WHICH Function, 370
WHOLE, 290
WIDTH, 161
WILD, 110, 110
WILD index filter, 110
WORKUNIT, 67, 371
Workunit, 74
WORKUNIT Function, 371
Workunit OUTPUT, 287
WUID, 371

X

XML, 71, 73, 278, 283, 285, 289, 295
XML Files, 283
XMLDECODE, 372
XMLDECODE Function, 372
XMLDEFAULT, 60
XMLENCODE, 373
XMLENCODE Function, 373
XMLPROJECT, 102
XMLTEXT, 102
XMLUNICODE, 102
XOR Operator, 32
XPATH, 60, 228, 336