verify: Verify Rule Check System Reference

CML00061-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

January 19, 2018

Contents

1 Introduction

The Code Magus Limited `verify` Rule Check System is used for mass checking cases or scenarios against a defined rule configuration and runs on the z/OS, Windows and Unix/Linux platforms. The scenarios or cases requiring checking are kept in a container, database or file system, whose access method, object identifier and options can be described by a `recio` open specification string (or open spec). Refer to recio: Record Stream I/O Library Version 1 [1] for details on the `recio` open spec strings and to [2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16] for details on the various access methods available. Typically, all the data required to be checked as a case item by `verify` are presented in a single record. This record would, for example, contain the relevant state of a system before a change, the transaction that initiated the state change, and the relevant state of a system after a change. In `verify` these records are mapped using the `objtypes` library as described in objtypes: Configuring for Object Recognition, Generation and Manipulation [6] and associated application or database metadata. Fields can further be used in expressions as described in section 5 on page 34.

Parameterisation of configurations of `verify` is catered for using the `applparms` library and associated interfaces as described in applparms: Application Parameters Library User Guide and Reference Version 1 [14].

The script engine used in `verify` uses the `debugapi` debugger interface described in debugapi: Debug API User Guide and Reference Version 1 [17]. This interface allows remote and local debugging sessions, and also allows for integration of the script engine into an IDE that supports a debugging graphical user interface.

2 Verify Processing

Figure 1 illustrates the components of `verify` from a processing point of view. The components of `verify` are discussed in this document and the cited references; however a description of `verify` processing involves more than the internal components of the system. This section describes `verify` processing from the point of view of the inputs and outputs.

There are a number of inputs to the execution of `verify`:

- **Configuration:** An item that is introduced by definition in the `verify` configuration file must be defined before (closer to the start) the use of or reference to the defined item in a later defined item.
  - **Files:** Bindings in the form of recio open spec strings for the input file that contains the cases to be processed; as well as the open spec strings for the output files to which records are copied as they exit the filtering and test process at various stages.

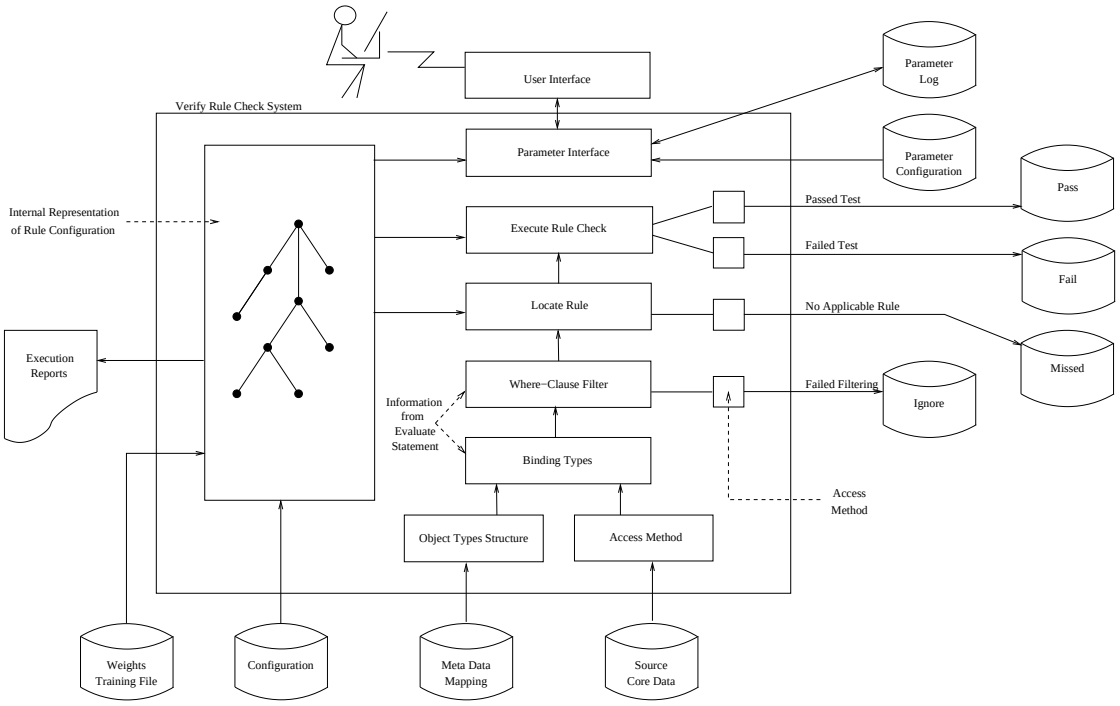


Figure 1: `verify` processing showing internal structure and interfaces

- **Meta-data:** Bindings to the object file definition which describes the layout of the input records being processed and, in particular, to the type of record that should be processed by the `verify` configuration.
- **Class conditions:** Definitions of the conditions used to assign the data into various classes in the various dimensions of the application data space. Each condition is defined with a name, a title and a predicate over the symbols of the input record mapping as well as any prior or pre-defined global variables. An item belongs to the class corresponding to the condition if the corresponding predicate evaluates to true.
- **Supporting Functions:** Named functions with the formal parameters and corresponding types, the function return type and the function body make up the definition of a script supporting function.
- **Supporting Procedures:** Procedures are called at various points to initiate processing. For example, on determining that a various rule applies to a case, that case is evaluated by invoking the corresponding procedure. Also, procedures can be invoked when certain pre-defined events occur during the execution of `verify`.
- **Rules:** Named rules are used to determine whether or not a case time passes or fails. A rule has a name and a title for identification and description; a list of the class conditions that when evaluated to true (that is, the item appears in

each of the classes corresponding to the named conditions) indicates that the rule is applicable to the item; and the name and parameters of the procedure to apply to determine whether the item satisfying all the requisite condition classes passes or fails the test.

- **Reports:** Additional user defined reports and be defined and these can be associated with events. When associated with an `event`, a report detail line is evaluated and written to the report at the point that the event is raised. If an event also has an associated `call` procedure clause, then the associated procedure will be called before the report detail line is prepared.

  Reports do not have to be associated with events. A report definition can be referenced by a *Report*-statement. A *Report*-statement is an executable statement which refers to a report. The execution of a *Report*-statement causes a report detail line to be produced at the time of execution of the statement.

- **Events:** In addition to the procedures used as the bodies for the rule check logic, procedures may also be invoked at various predefined points within the execution of `verify`. The trigger for these procedure invocations are known as *events*. A report may also be associated an event. If a report is associated with an event then a report detail line is generated when the event occurs.

- Source File: The input file that contains the cases to process during an instance of execution of `verify` is bound to within the configuration file. This binding is in the form of an open spec string and hence includes the access method name, object name, and any application access method option name-value pairs.
- Training File: The result of sampling the input data against the input source file (or a subset of the input source file) is a training file of weights. This training file is read in again during the normal execution of `verify` and is used to determine the condition checking priority order when building the decision tree.

The execution of `verify` with a configuration against a set of cases produces a number of outputs. The items that actually have a rule applied are those that pass certain filtering as the input file is processed. The `source` file is processed sequentially, each record read from the input file is checked to make sure that it matches the type named on the `evaluate` statement. If the record matches the type, then an optional predicate where-clause on the `evaluate` statement is tested if present. If the where-clause predicate evaluates to true, then the record is considered for testing. The decision tree is applied to all records considered for testing. If this yields a process which can be applied to the record, then that process will evaluate the case and determine whether the case passes or fails.

Normal execution of `verify` produces the following outputs:

- **Tested File:** Every record read from the `source` file that passes the `evaluate` statement object type name and optional where-clause predicate is written to the `tested` file if a binding is present. This represents all records passed into verify for testing. Note this does not mean that the record will actually find its way to a test procedure under a rule, it just means that the decision tree will be consulted in order to determine a rule for the testing of the record.
- **Ignore File:** If a binding is present for the `ignore` file and if the item does not pass the `evaluate` statement object type or fails the `evaluate` where-clause if present, then the record is not passed on for testing and is written to the `ignore` file. Further if the record is passed on for testing and a rule is found to apply, then if the rule executes the `skip` statement then in this case the record is written to the `ignore` file. In addition, when the rule executes the `skip` statement the count of the number of times the rule is applied is not incremented.
- **Missed File:** If a record passed for testing (a candidate for writing to the `tested` file, if present), but that `verify` fails to determine a rule for the record, then the record is written to the `missed` file. The `missed` file is mandatory.
- **Passed File:** If a case record is passed in for testing, a rule applicable to the case is found and that rule determines that the case passes the rule, then the case is written to the `passed` file. The `passed` file is mandatory.
- **Failed File:** If a record is passed in for testing, a rule applicable to the case is found, but that rule determines that the case fails, then the case is written to the `failed` file. The `passed` file is mandatory.
- **Report File:** The result of the execution of `verify` on the supplied inputs produces a report file which summarises the instance of the execution of `verify`. In addition, the report file includes trace and decision data as well as formatted items to support the maintenance of the rules. The report file also contains the script output written as the result of `print` and `debug` statements executed.
- **Decision Tree:** Once the structure of the decision tree has been determined, it is possible to obtain a graphic rendering of the decision tree showing the conditions on the path to the rule nodes. Figure 2 on page 7 shows an example of what this would look like for a few of the rules and conditions from a sample configuration. There is a tool for the interactive navigation of the rendered graph.

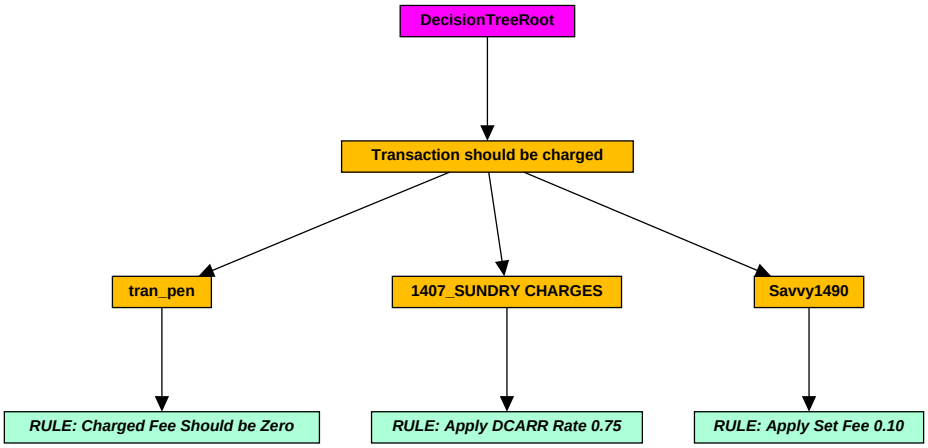


Figure 2: Graphical rendering of dynamically built decision tree

3 Processing

The process of determining the rule to apply to a case involves determining within which class, within specific dimensions that case belongs. For example, gender might be a dimension which partitions the data into male and female classes. We use the term dimension here, rather than the term attribute as the class may be represented by more than one attribute of the underlying data (an out of product rule condition could, for instance, be determined by a high-balance and the recent history of a particular transaction type). In verify, we define a class by a `condition` and we expect those conditions that form part of the same dimension to form a partition of the data, or at the very least, to uniquely assign every item to only one class within a dimension.

There could be any number of dimensions in the case data, with each dimension expecting to possess the property described above. For the sake of illustration, however, we assume two dimensions as depicted in Figure 3 on page 8.

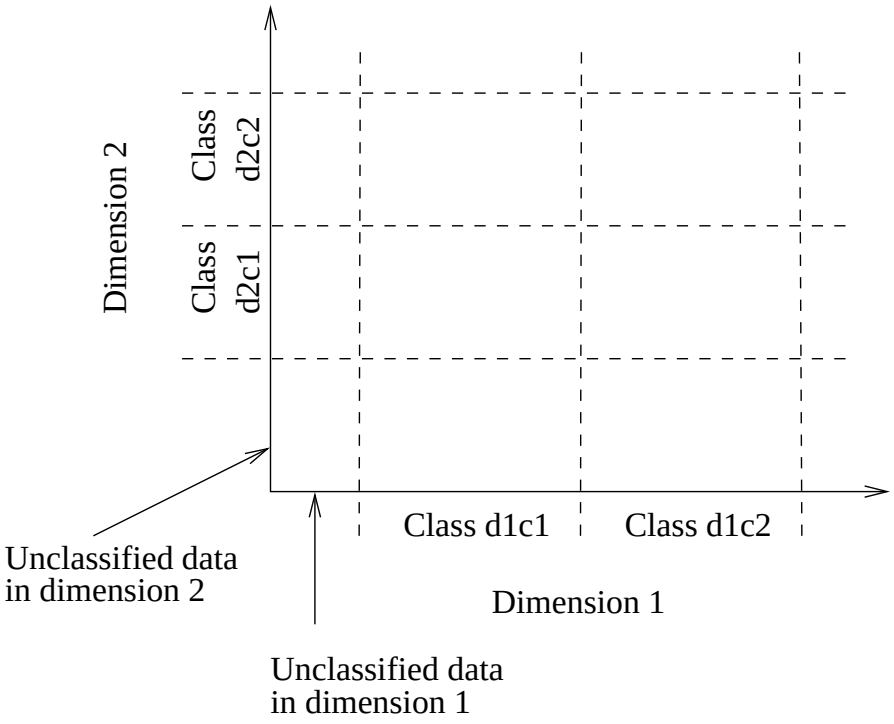


Figure 3: Depiction of two dimensions through the case data showing two classes for each dimension

In this example, there are two classes within each dimension. If each of the axes depict the full range of possible case data, then each of the dimensions depicts ranges on the axes of unclassified data (and hence, in this example, the dimensions do not form partitions of the data).

A rule in `verify` consists of the predicated conditions (alternatively, rule requirements or preconditions), together with the means of determining whether a case passes or fails

(this is in the form a procedure that must be invoked to determine this). As a side effect, the script code that performs the evaluation may maintain global data elements for `verify` to report on (this allows, for example, valuations on all the classified cases to be performed and reported on, or for test runs over the data to be done so that what-if scenarios can be assessed).

There is no limit to the number of dimensions, and within each dimension there is no limit to the number of possible classes; hence there is no limit to the number of conditions that one might require. With a larger number of dimensions and a large number of classes, the investment in an evaluation order might be significant. Also, the effect of the evaluation order impacts the efficiency of the process of determining which rule should apply, consequently verify employs a scheme whereby the case data is sampled, and then the information gleaned from the sampling is used to decide the structure of the rule determination decision tree. The result of this is that there is no expensive manual optimisation required; and should the classification of the data change significantly, a simple sampling of the data will result in the re-establishment of an optimal evaluation order decision tree.

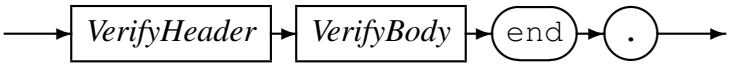
In terms of the example in Figure 3 on page 8, if it is determined that condition `d1c2` is more frequent than condition `d2c1`, then this will cause that condition to be tested earlier by placing it higher up in the decision tree. Also, a dimension which is significantly lopsided toward a particular class should have that condition checked as soon as possible.

Note that the above does not change the rule evaluation order. Rules have their conditions added to the paths of the decision tree in the sample determined priority order. This means that default rules can be used, and which just need to follow the higher priority rules in the configuration file.

4 verify Configuration

A verify configuration is defined in an application specific grammar which is described in this section. There is a broad structure to the configuration file and in general an item needs to be defined before it can be referenced within the configuration. This is to support the one-pass compiler of the verify configuration.

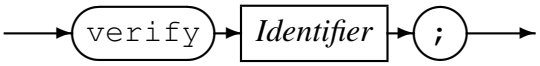
VerifyConfig



A verify configuration file comprises a header, a body and is terminated by the keyword end followed by a full stop.

4.1 Configuration File Header

VerifyHeader



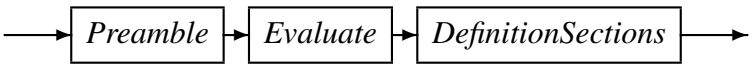
The header of a configuration file identifies it as a verify configuration and gives the configuration an internal name.

The following is an example of the *VerifyHeader*:

```
verify CurrentAccounts;
```

4.2 Configuration File Body

VerifyBody



The body of a verify configuration in turn, comprises three sections: A *Preamble* for definitions and environment bindings such as files to process, parameter configurations, and options; an *Evaluate* section which contains the evaluate statement and marks the end of the *Preamble*; and the *DefinitionSections* which define the conditions, rules, processes and events required for the processing of the cases files.

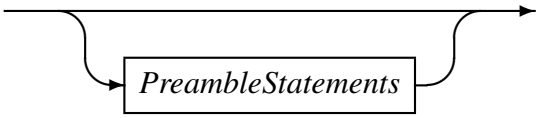
Logically, the file is split into sections which describe the processing environment (the *Preamble* section); a section to describe what is to be processed (the *Evaluate* section); and a section to describe how this is to be processed.

4.2.1 Configuration Preamble

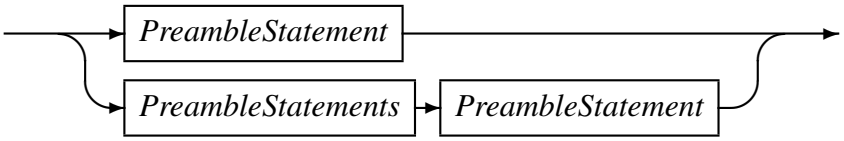
The *Preamble* section of a `verify` configuration describes the bindings to the environment in which the processing is to take place.

Preamble

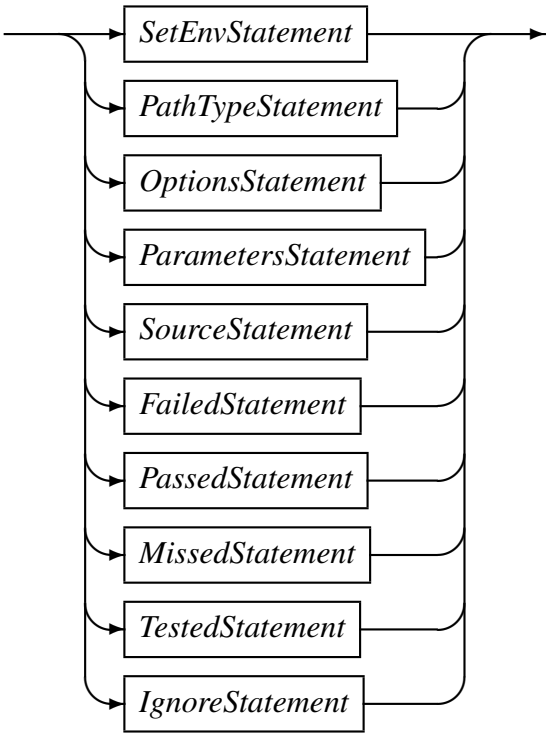
OptPreambleStatements



PreambleStatements



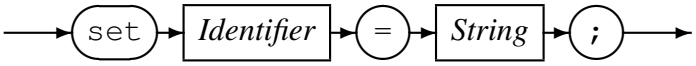
PreambleStatement



There are a number of different types of statements that make up the *Preamble* section. The *Preamble* section follows the *VerifyHeader* and is terminated by the *Evaluate* section. The statements of the *Preamble* section may appear in any order (again, only with

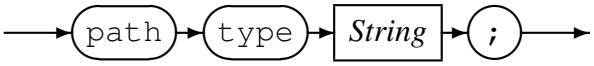
the restriction that any references in a *Preamble* statement must follow the statement that defines the item being referenced).

SetEnvStatement



The *SetEnvStatement* sets the environment variable corresponding to the *Identifier* to the value of the result *String* (this is the value obtained by resolving all environment variables and concatenating adjacent strings).

PathTypeStatement

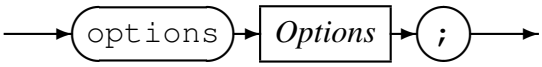


The *PathTypeStatement* provides a string pattern which is used for resolving object types file names. The string is expected to contain a sequence of characters suitable for the s-type conversion specifier as used in the `sprintf(3)` function.

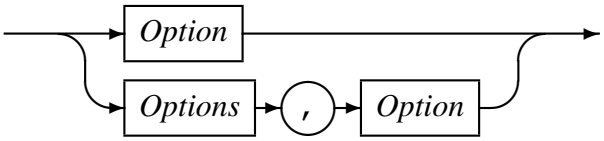
The following is an example of the *PathTypeStatement* in which the environment variable `TYPEHOME` is expected to return the directory path containing the `objtypes` files:

```
path type ${TYPEHOME} "%s.objtypes";
```

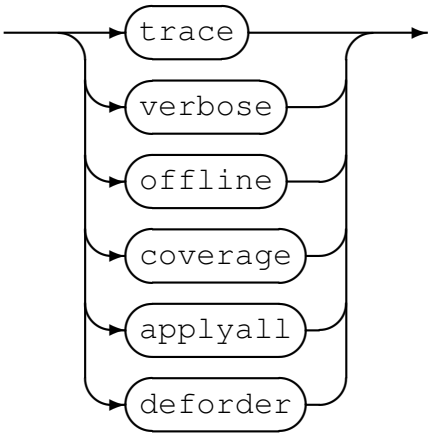
OptionsStatement



Options



Option



The *OptionsStatement* is used to set certain overriding flags for processing. The `trace` option indicates that detailed trace information is to be recorded during the execution of the rule-checking.

The `verbose` option indicates that all processing of embedded artifacts such as access method definitions, object types definitions, copybooks, and application parameter definition files should be expanded in full. Additionally, any component that supports a verbose processing mode is expected to have this enabled.

The `offline` processing mode indicates that the application parameter configuration is not to interact with the user, and if all parameters are successfully set then processing will continue as normal. If for some reason the parameters cannot be satisfied by the current values, then processing terminates with an error message indicating the intervention required. If this option is not in effect, then the execution of `verify` is assumed to be interactive and a user interface for application parameters is presented to the user.

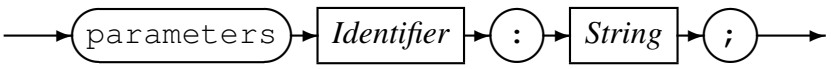
The `coverage` processing option is a request to produce coverage statistics for the rules that were applied, passed and failed. The report is produced once all the cases have been processed.

The `applyall` processing mode attempts to apply all rules that are applicable to each of the buffers. This default mode is to attempt to apply only the first rule which is applicable to a buffer.

The `deforder` processing mode selects rules to apply in the order that they are defined in the configuration file. The default mode is to attempt to check for rules applicable in the weighted-tree optimised order. This option allows the order of rules to be checked for applicability in an order defined by the user configuring the rule set.

```
options trace, verbose;
```

ParametersStatement



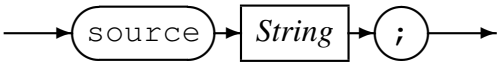
The *ParametersStatement String* identifies an `applparms` library [14] definition file. This file defines the parameters that need to be included in the defined global variables when `verify` executes the configuration. The value of the *Identifier* is used to qualify the variables defined in the given `applparms` definition file. There can be any number of configuration files attached in this manner, each requiring their own *ParametersStatement* (note that an `applparms` configuration is intended to satisfy all the parameter requirements of an application or solution instance, and hence considered normal that there is only one `applparms` configuration for each `verify` configuration).

Note that all parameters introduced by a *ParametersStatement* have a type of `string`.

The following is an example of a *ParametersStatement* in which the `applparms` parameters defined in the `CASATEST.apd` configuration will be known to the `verify` as global variables qualified by `MyVars` (for example, `MyVars.CACHFeeCap`):

```
parameters MyVars : "CASATEST.apd";
```

SourceStatement



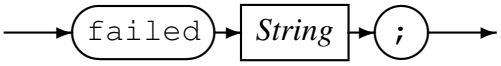
The *SourceStatement* binds the `source` file using the `recio` [1] supplied *String* as the open spec string (this is the value obtained by resolving all environment variables and concatenating adjacent strings). The open spec string must be suitable for opening and reading the file in a sequential input manner and is the file that supplies the cases to be filtered and tested.

There must be exactly one `source`-statement in the configuration file.

The following is an example of a *SourceStatement* in which the open spec string is expected to be supplied by the environment variable `SOURCE_SPEC`:

```
source ${SOURCE_SPEC};
```

FailedStatement



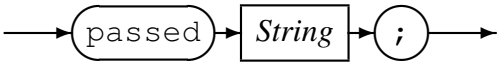
The *FailedStatement* binds the `failed` file using the `recio` [1] supplied *String* as the open spec string (this is the value obtained by resolving all environment variables and concatenating adjacent strings). The open spec string must be suitable for opening and writing the file in a sequential output manner and is the file to which all cases are written that have passed filtering, have successfully located a rule, have had the rule applied, but the rule determined that the case has failed.

There must be exactly one `failed`-statement in the configuration file.

The following is an example of the *FailedStatement*:

```
failed "binary(" ${FAILED_NAME} ",recfm=f,reclen=654,mode=wb)";
```

PassedStatement



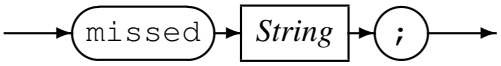
The *PassedStatement* binds the `passed` file using the `recio` [1] supplied *String* as the open spec string (this is the value obtained by resolving all environment variables and concatenating adjacent strings). The open spec string must be suitable for opening and writing the file in a sequential output manner and is the file to which all cases are written that have passed filtering, have successfully located a rule, have had the rule applied, and the rule determined that the case has passed.

There must be exactly one `passed`-statement in the configuration file.

The following is an example of the *PassedStatement*:

```
passed "binary(" ${PASSED_NAME} ",recfm=f,reclen=654,mode=wb)";
```

MissedStatement



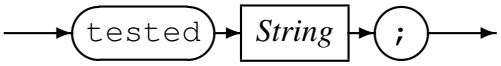
The *MissedStatement* binds the `missed` file using the `recio` [1] supplied *String* as the open spec string (this is the value obtained by resolving all environment variables and concatenating adjacent strings). The open spec string must be suitable for opening and writing the file in a sequential output manner and is the file to which all cases are written that have passed filtering, but have not been successful in locating a rule.

There must be exactly one `missed`-statement in the configuration file.

The following is an example of the *MissedStatement*:

```
missed "binary(" ${MISSED_NAME} ",recfm=f,reclen=654,mode=wb)";
```

TestedStatement



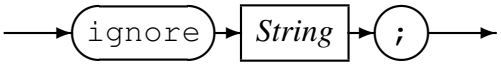
The *TestedStatement* binds the `tested` file using the `recio` [1] supplied *String* as the open spec string (this is the value obtained by resolving all environment variables and concatenating adjacent strings). The open spec string must be suitable for opening and writing the file in a sequential output manner and is the file to which all cases are written that have passed filtering.

There must be at most one `tested`-statement in the configuration file.

The following is an example of the *TestedStatement*:

```
tested "binary(" ${TESTED_NAME} ",recfm=f,reclen=654,mode=wb)";
```

IgnoreStatement



The *IgnoreStatement* binds the `ignore` file using the `recio` [1] supplied *String* as the open spec string (this is the value obtained by resolving all environment variables and concatenating adjacent strings). The open spec string must be suitable for opening and writing the file in a sequential output manner and is the file to which all cases are written that have not passed filtering.

There must be at most one `ignore`-statement in the configuration file.

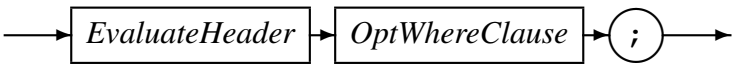
The following is an example of the *IgnoreStatement*:

```
ignore "binary(" ${IGNORE_NAME} ",recfm=f,reclen=654,mode=wb)";
```

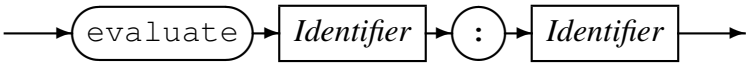
4.2.2 Configuration Evaluate Statement

The `evaluate`-statement is mandatory and is used to specify the meta-data binding to the records of the `source` file; and which subset of these records are to be processed by matching the records to the particular object type; and an optional predicate over that record type that must be true in order for the record to be considered for testing. This process of choosing which records in the input `source` file to process is termed *filtering* in this document.

Evaluate

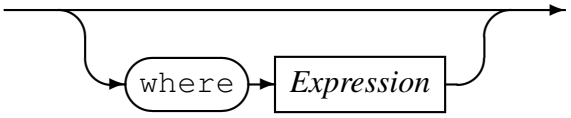


EvaluateHeader



The *Identifier* on the left of the colon is the name of the object types collection that should be used. The full path name of this file is obtained by inserting the value of this *Identifier* into the string specified by the *PathType* statement. The *Identifier* on the right of the colon is the name of the object type within the object types file that should be used for filtering the records of the `source` file. This *Identifier* is also used as the qualifier of all the symbols defined by the chosen object type.

OptWhereClause



Further filtering of the records is permitted using an optional `where`-clause to the *Evaluate* statement. If a `where`-clause is present, then the record passes filtering if it is of the correct object type and the predicate evaluates to true. The `where`-clause predicate is defined in terms of symbols over the selected object type and any global symbols that have previously been defined in the configuration file.

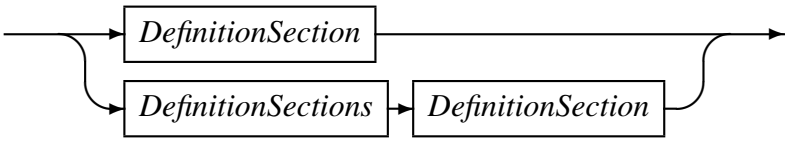
The following is an example of the `evaluate`-statement using a `where`-clause. In this example, the `where`-clause predicate refers to the predefined symbol `source_count` which is a counter of the number of `source` file records read. Given the previous example *PathTypeStatement*, and assuming the environment variable `TYPEHOME` has the value `/home/testdata/CodeMagus/objtypes/"`, the object types file referred to in the `evaluate`-statement would then be `/home/testdata/CodeMagus/-objtypes/JOINRECD.objtypes`; and an object type called `test_case` is expected to be defined in this file.

```
evaluate JOINRECD:test_case where source_count < 5;
```

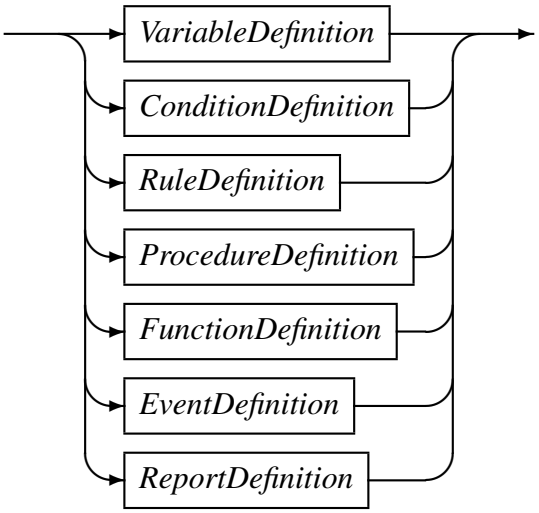
4.2.3 Configuration Definitions Sections

The remainder of the *VerifyBody* of the `verify` configuration file comprises definition sections of various types. The definition sections all influence the processing of `source` cases file in various ways. The *DefinitionSections* may appear in any order, but if a *DefinitionSection* refers to an element introduced by another *DefinitionSection*, then that referring *DefinitionSection* must follow the defining *DefinitionSection* in the `verify` configuration file.

DefinitionSections



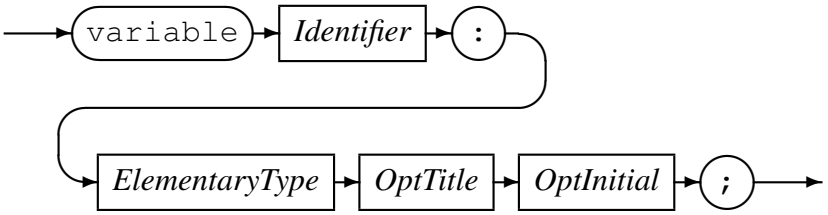
DefinitionSection



4.2.4 Variable Definition

A *VariableDefinition* introduces a global variable into to the `verify` configuration.

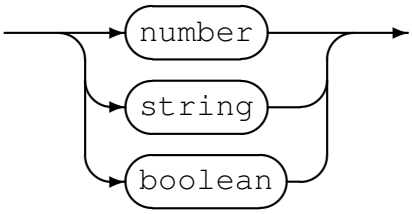
VariableDefinition



The variable named as the *Identifier* is defined at the point of processing the configuration file where the *VariableDefinition* statement is found, and is available to be refer-

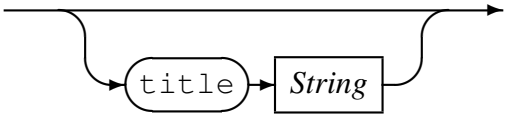
enced in further *DefinitionSections* following the *VariableDefinition*. A global variable, as with any variable within `verify`, is typed as an elementary typed item. Optionally, a global variable may have a descriptive title (used in reporting) and an optional initial value.

ElementaryType



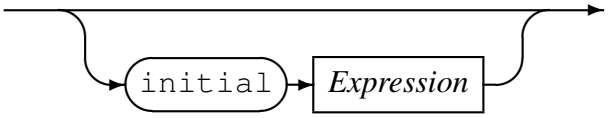
There are only three elementary types: *Numbers*, *Strings* and *Booleans*. A type of number indicates that the item may contain any decimal, fixed point or floating point value. *Numbers* have 32 significant decimal digits. A *String* comprises a sequence of characters of graphic characters. The encoding of the *Strings* is intended to be hidden from the `verify` script code, and the actual encoding of the data is taken into account by the object types definition file. A *Boolean* item is truth valued and may have one of two values true or false (there are predefined variables `true` and `false` with fixed values of true and false respectively).

OptTitle



An optional `title`-clause of the defined global variable is intended to provide a fuller explanation of the variable and is used in reporting and tracing.

OptInitial



An optional `initial`-clause is an expression over previously defined symbols with values which is evaluated and assigned to the global variable at the point at which it is defined. The expression must evaluate to a type that is the same as the type of the variable.

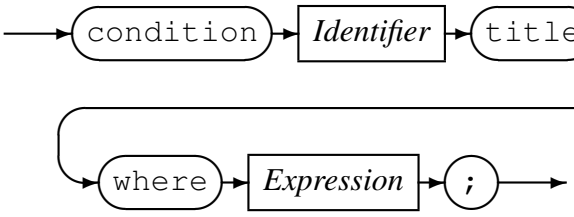
The following is an example of a *VariableDefinition*:

```
variable GrossFeeAmount : number
   title "Sum of all fees charged for"
   initial 0.00;
```

4.2.5 Condition Definition

A condition-definition defines a class within a dimension of the data. The class is defined by a condition-predicate and all source file items belong to the class if the condition-predicate evaluates to true in the context of that item.

ConditionDefinition



The condition with a name indicated by the *Identifier* is defined at the point that the condition-statement is encountered within the verify configuration. Associated with a configuration is a title-clause which is expected to give a fuller explanation of the condition and which can be used in reporting and tracing. The where-clause *Expression* is a boolean valued predicate that determines class membership.

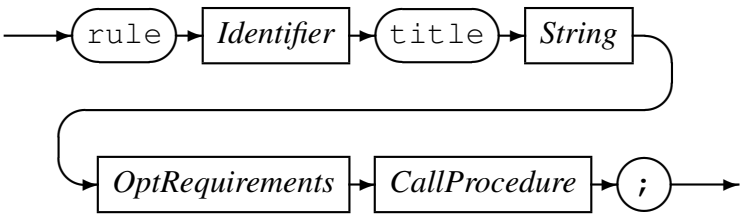
The following is an example of a *ConditionDefinition*:

```
condition Age_55_and_over
   title "Account holder is 55 years or older"
   where (test_case.JOIN_RECORD.JOIN_AGE_INDICATOR = "M")
      or (test_case.JOIN_RECORD.JOIN_AGE_INDICATOR = "O");
```

4.2.6 Rule Definition

A rule defines the actions to be performed, when an item is determined to belong to it, if a series of conditions are true for that item. It is the responsibility of the rule to prepare for a verify event to call a case item a pass or fail; or to directly declare the case item a pass or fail.

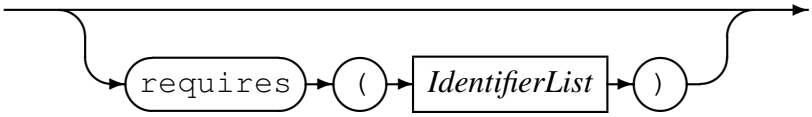
RuleDefinition



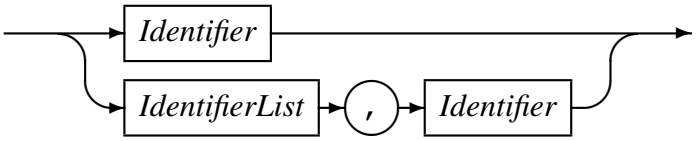
The name of a rule is supplied by the value corresponding to the *Identifier*. A *rule* also has a title which is expected to have a fuller explanation of the rule and is used in reporting and tracing. The *OptRequirements* lists the *condition* for the applicability of the rule to the current source case item. A default rule, defined as the last rule in

a verify configuration file, may be defined without any conditions and would act as a catch all rule, applicable if no other rule applied to an item. Associated with every rule is a *CallProcedure*-clause which provides the name and actual parameter list of the procedure to call to evaluate (or initiate the evaluation) of the case item.

OptRequirements



IdentifierList



Rules are tested against each item in the order that they appear in the verify configuration file. And the first rule that has all of its *Requirement* conditions satisfied within the context of the item becomes the rule applicable to the item. At most one rule is applicable to an item.

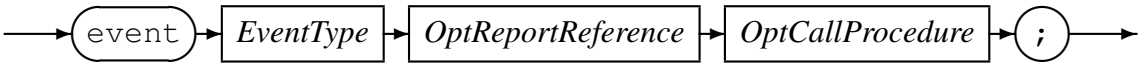
The following is an example of a *RuleDefinition*:

```
rule should_charge_1486_CHARGE_UNPAID_ITEM_Dezign_Student
   title "should_charge_1486_CHARGE_UNPAID_ITEM_Dezign_Student"
   requires (Dezign_Student, TC_1486_CHARGE_UNPAID_ITEM,should_charge)
   call set_fee(0.75);
```

4.2.7 Event Definition

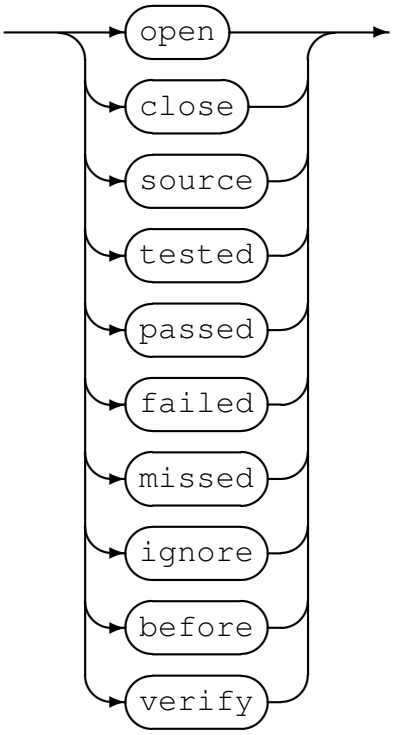
There are various points within the verify processing where an implicit call to a procedure is allowed. Where appropriate, these points in the processing are made within the context of the current item being processed.

EventDefinition



An event does not have a name. Rather, an event has an *EventType* which indicates the point in processing that indicates when the *CallProcedure* will be invoked. There is a fixed list of *EventTypes* corresponding to specific points during the execution of verify.

EventType



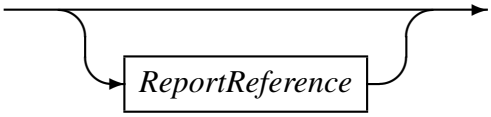
- `open`: The *CallProcedure* defined for this event is invoked when the `verify` configuration is successfully opened.
- `close`: The *CallProcedure* defined for this event is invoked when the `verify` configuration is closed, provided that the configuration was successfully opened.
- `source`: The *CallProcedure* defined for this event is invoked whenever record is read from the `source` file.
- `tested`: The *CallProcedure* defined for this event is invoked whenever a record passes filtering. The context of the call includes the record just read.
- `passed`: The *CallProcedure* defined for this event is invoked whenever a record is written to the `passed` file. The context of the call includes the record written.
- `failed`: The *CallProcedure* defined for this event is invoked whenever a record is written to the `failed` file. The context of the call includes the record written.
- `missed`: The *CallProcedure* defined for this event is invoked whenever a record is written to the `missed` file (that is when a rule cannot be located for the item). The context of the call includes the record written.
- `ignore`: The *CallProcedure* defined for this event is invoked whenever a record is written to the `ignore` file (that is, when the item fails the filtering).
- `before`: The *CallProcedure* defined for this event is invoked whenever a rule

has been successfully identified for the case, but before the *CallProcedure* of the `rule` has been invoked. The context of the call includes the current item.

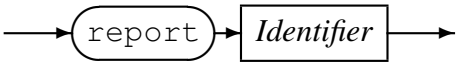
- `verify`: The *CallProcedure* defined for this event is invoked whenever a rule has been successfully identified for the case, and after the *CallProcedure* of the `rule` has been invoked. The context of the call includes the current item.

4.2.8 Report Definition

OptReportReference

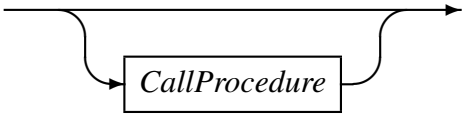


ReportReference

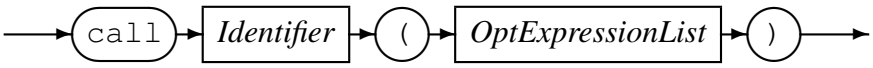


ReportReference-clauses are used in *EventDefinitions* in order to associate the event with a *ReportDefinition*. During execution, if an event is associated with a report, then the triggering of that report will cause a detail line to be added to the defined report. The preparation and formatting of items for a report line when triggered by the occurrence of an event takes place after the execution of the procedure identified in the *CallProcedure*-clause, if present.

OptCallProcedure

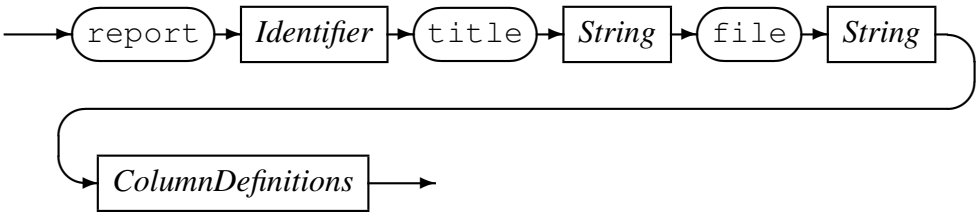


CallProcedure

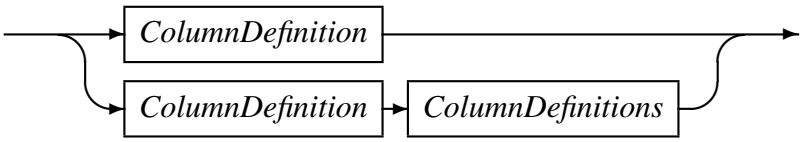


CallProcedure-clauses are used in *EventDefinitions* and *RuleDefinitions*. The *Identifier* in a *CallProcedure*-clause names a defined `procedure`; and the actual parameters defined in a *OptExpressionList* are expected to evaluate to types matching that defined `procedure`.

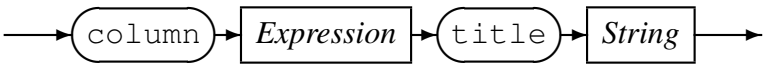
ReportDefinition



ColumnDefinitions



ColumnDefinition

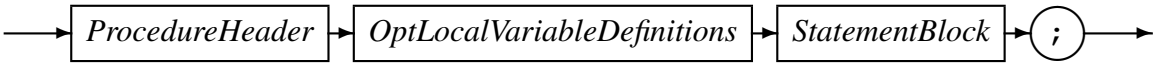


A *ReportDefinition*-statement defines a single report in `verify`. Reports are defined with a name and a title. The name is used within the `verify` configuration file to refer to the report when indicating at which points content should be added to the report. Content can be added to a report by associating a report with an event.

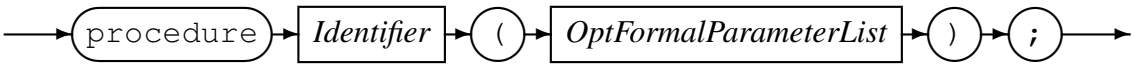
In addition to the report name and title, a `recio` open specification string is also associated with a report. This is used to define the file that the report will be written to.

The content of a report is defined by the listing the columns that are to appear in the report. This defines both the column headings as well as the report line details. Column headings are defined as literal *Strings* in the `report column` definition; and the content of the report detail lines are defined as corresponding *Expressions*. These *Expressions* are evaluated within the global scope of the `verify` configuration.

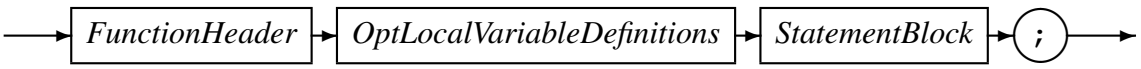
ProcedureDefinition



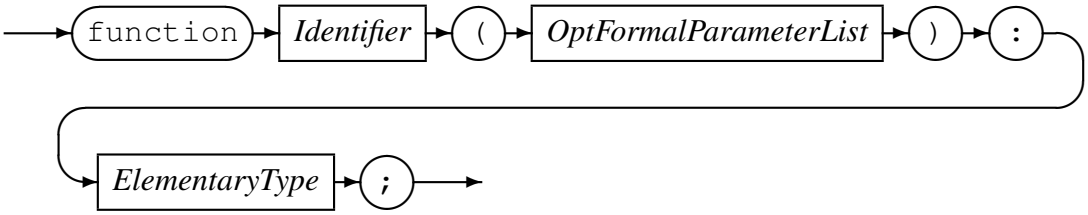
ProcedureHeader



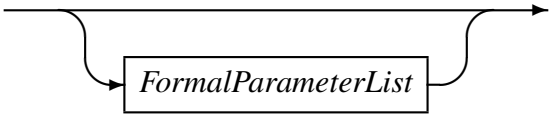
FunctionDefinition



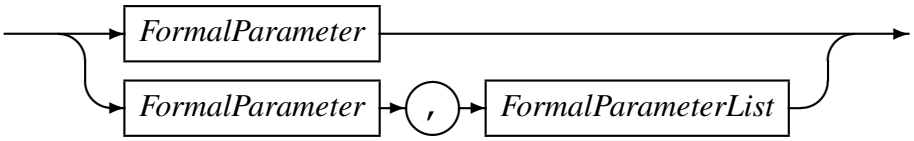
FunctionHeader



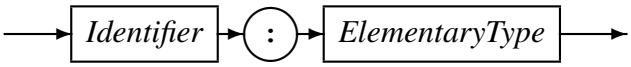
OptFormalParameterList



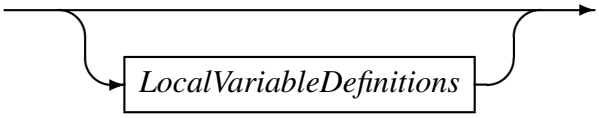
FormalParameterList



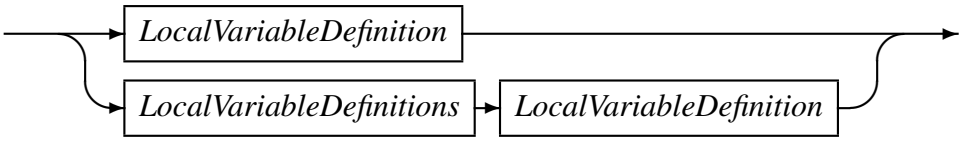
FormalParameter



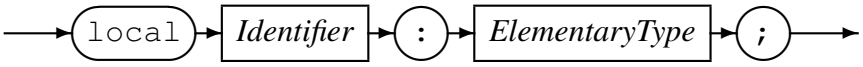
OptLocalVariableDefinitions



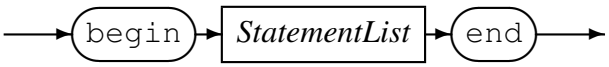
LocalVariableDefinitions



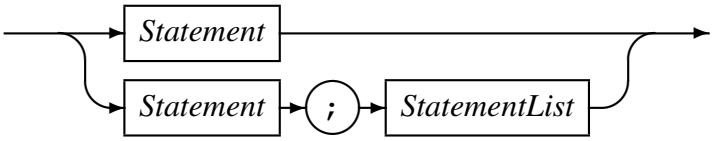
LocalVariableDefinition



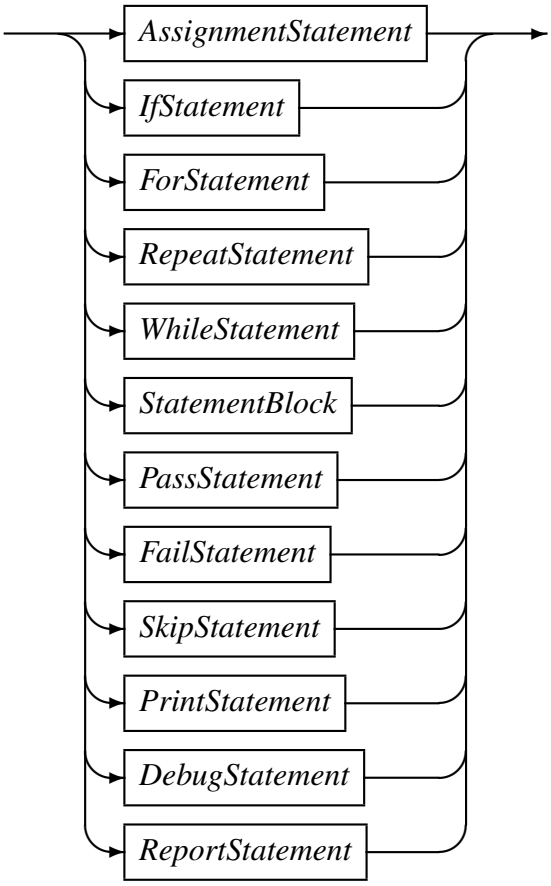
StatementBlock



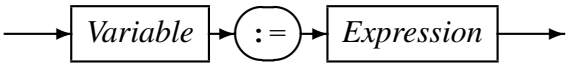
StatementList



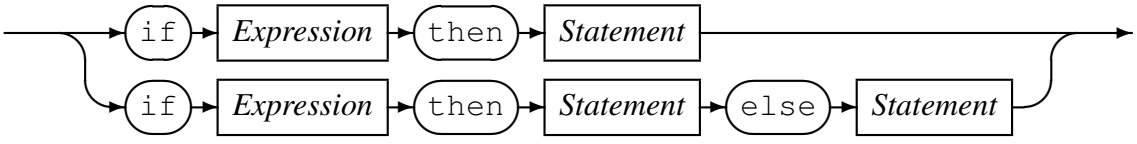
Statement



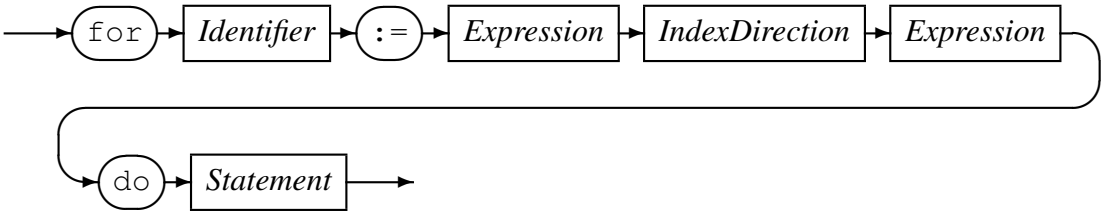
AssignmentStatement



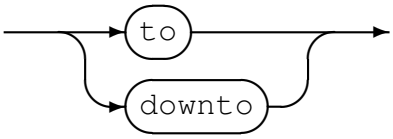
IfStatement



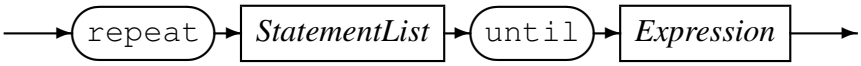
ForStatement



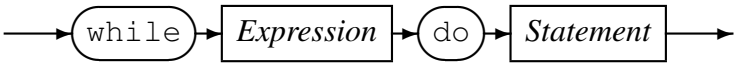
IndexDirection



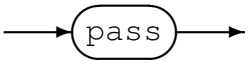
RepeatStatement



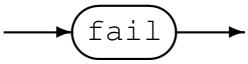
WhileStatement



PassStatement

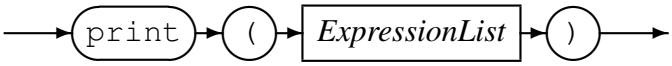


FailStatement

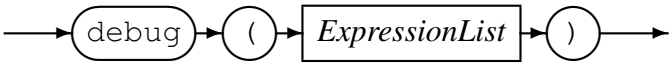


SkipStatement

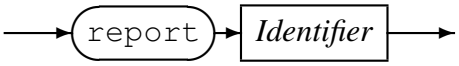
PrintStatement



DebugStatement



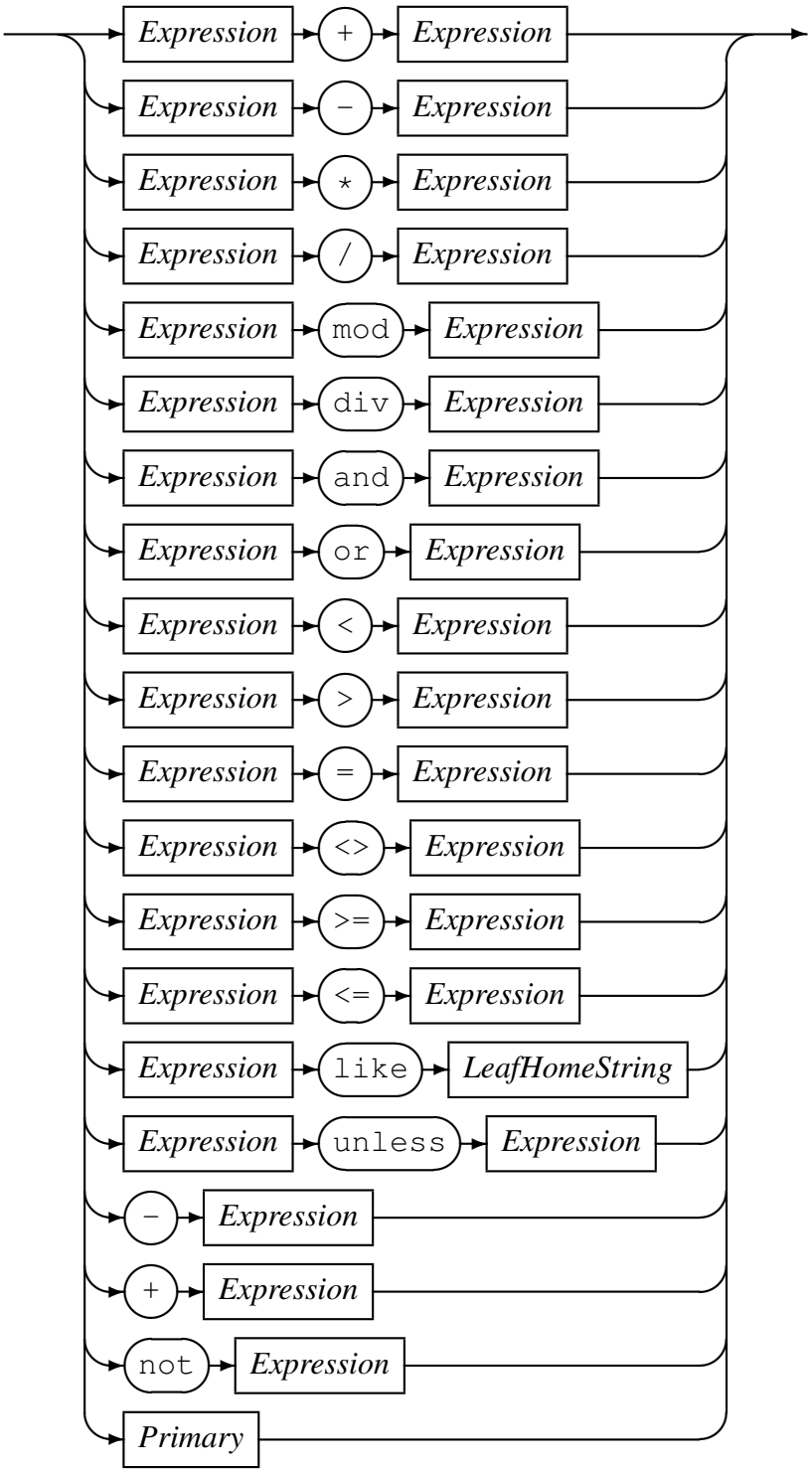
ReportStatement



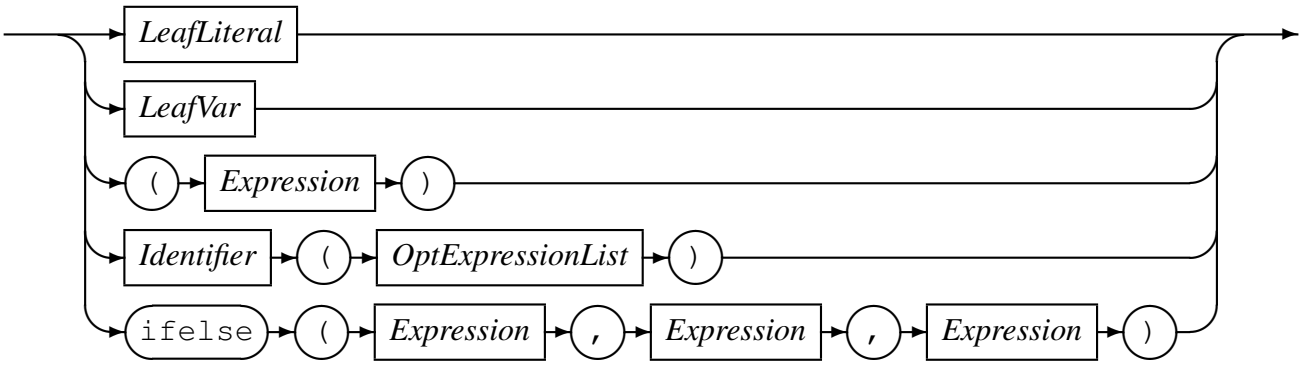
The *ReportStatement*-statement refers to a *ReportDefinition* by name. The execution of a *Report*-statement causes the columns of the associated report definition to be eval-

uated and the resultant detail line to be added to the report file. Any number of *Report*-statements and *EventDefinitions* may refer to the same report. A report line will be generated for each *Report*-statement executed and for each triggered *Event* which contains a reference to the report.

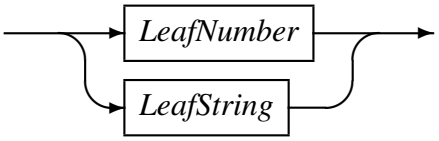
Expression


Expression + Expression
Expression - Expression
Expression * Expression
Expression / Expression
Expression mod Expression
Expression div Expression
Expression and Expression
Expression or Expression
Expression < Expression
Expression > Expression
Expression = Expression
Expression <> Expression
Expression >= Expression
Expression <= Expression
Expression like LeafHomeString
Expression unless Expression
- Expression
+ Expression
not Expression
Primary

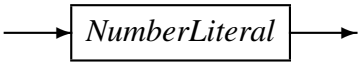
Primary



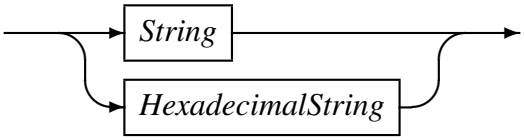
LeafLiteral



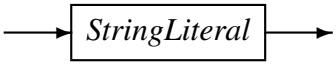
LeafNumber



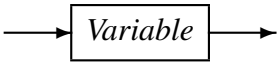
LeafString



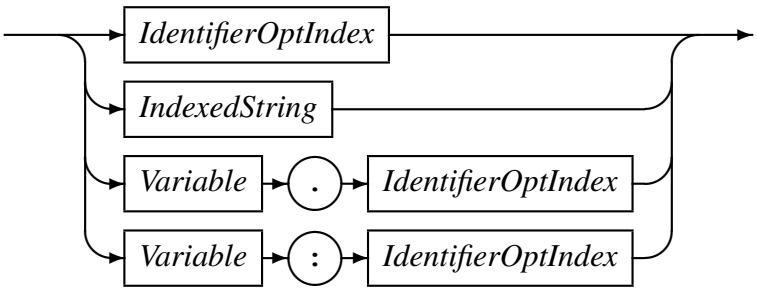
LeafHomeString



LeafVar



Variable



A *Variable* may be user defined or pre-defined. User defined variables are introduced in the *VariableDefinition*, in *FunctionHeader*, in a *ProcedureHeader*, as a *LocalVariable* defined in a `function` or `procedure`, or as the result of a reference to an `objtypes` configuration for describing external data.

There are also a number of predefined variables:

Name	Type	Description
`true`	*Boolean*	Always has the value true
`false`	*Boolean*	Always has the value false
`source_count`	*Numeric*	A counter that is incremented each time that the source scenario file is read. This is whether or not the record is missed or ignored. The counting starts at 1, and after the last record is read from the source file, this variable contains the total number of records read.
`passed_count`	*Numeric*	A counter which is incremented each time an applied rule passes. Initially the value is zero. This value also keeps track of the number of records written to the passed file.
`failed_count`	*Numeric*	A counter which is incremented each time an applied rule fails. Initially the value is zero. This value also keeps track of the number of records written to the failed file.
`missed_count`	*Numeric*	A counter which is increment each time a record is missed and written to the missed scenarios file. Initially the value is zero. This value also keeps track of the number of records written to the missed file.
`tested_count`	*Numeric*	A counter which is incremented each time a record is considered for testing. Initially the value is zero. This value also keep track of the number of records written to the tested file.

continued on next page

continued from previous page

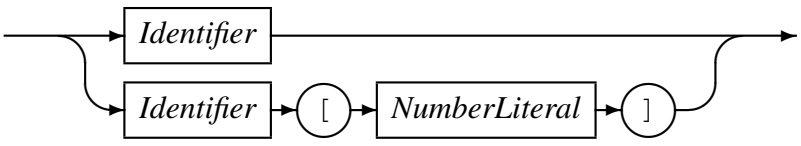
Name	Type	Description
ignore_count	*Numeric*	A counter which is incremented each time a record is found for which no rule is applicable. This value also keeps track of the number of records written to the ignore file.
current_rule_name	*String*	When a rule is found to be in context, either within the body of a rule, or within the scope of a before event, passed event, or a failed event, this variable will have the name of the rule in context. Outside of this context, the variable will still be defined, but will have a value indicating that no rule is currently in context.
current_rule_title	*String*	When a rule is found to be in context, either within the body of a rule, or within the scope of a before event, passed event, or a failed event, this variable will have the title of the rule in context. Outside of this context, the variable will still be defined, but will have a value indicating that no rule is currently in context.
current_rule_procedure	*String*	When a rule is found to be in context, either within the body of a rule, or within the scope of a before event, passed event, or failed event, this variable will have the name of the procedure of the rule in context. Outside of this context, the variable will still be defined, but will have a value indicating that no rule procedure name is currently in context.
current_rule_parmvalues	*String*	When a rule is found to be in context, either within the body of a rule, or within the scope of a before event, passed event, or failed event, this variable will have the names of the formal parameters and the values of the corresponding actual parameters formatted as a string. Outside of this context, the variable will still be defined, but will have a value indicating that no rule procedure parameter names and values currently in context.

continued on next page

continued from previous page		
Name	Type	Description
`diagnostic_path`	*String*	When a case is considered, a number of conditions may be checked against the current case. This process may or may not have found a corresponding rule. This variable is a trace variable which indicates the rule chosen as well as any conditions that have been checked against the current case. This condition detail includes, with the condition name, whether the condition was found to be true or false for the current case.
`diagnostic_path_true`	*String*	When a case is considered, a number of conditions may be checked against the current case. This process may or may not have found a corresponding rule. This variable is a trace variable which indicates the rule chosen as well as any conditions that have been checked and satisfied against the current case.
`diagnostic_path_false`	*String*	When a case is considered, a number of conditions may be checked against the current case. This process may or may not have found a corresponding rule. This variable is a trace variable which indicates the rule chosen as well as any conditions that have been checked and found not to be true against the current case.

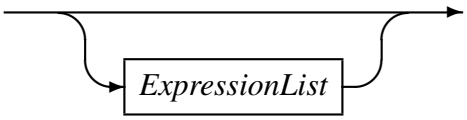
Table 1: Predefined Variables

IdentifierOptIndex

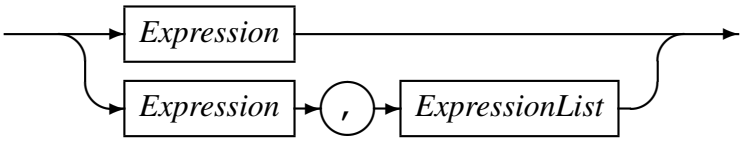


IndexedString

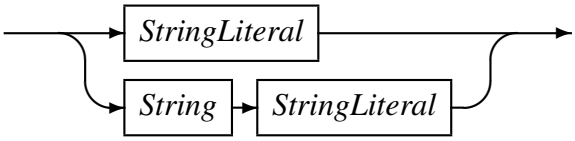
OptExpressionList



ExpressionList



String



5 Expression Evaluation

5.1 Expression Overview

The lexical elements of an expression are the variables, literals, operators and other character symbols used to form an expression. These lexical elements or tokens are separated by white spaces. White spaces include sequences of the space character, new-line character, the tab character and the linefeed character and their only function is to separate or delimit the tokens.

The lexical elements are often single characters having their own apparent meaning, but some are grouped together to form a word having a specific meaning. Included or associated with each token may be an attribute value.

An expression, made up of the constituent tokens into the syntax and semantics of the grammar, is then validated and evaluated by the expression evaluation library. The evaluation of an expression produces a value that can then be used within the context of the grammar of the specific Code Magus product within which it is specified.

Examples of expressions are:

1. `3+4`
2. `balance + 100`
3. `(account.balance >= 2000)`
4. `where (account.balance = 0)`
5. ```
   where (account.balance < 0 ) and
         (account.overdraft_facility = 'Y')
   ```
6. `SysString(account.balance)`

5.2 Expression Grammar

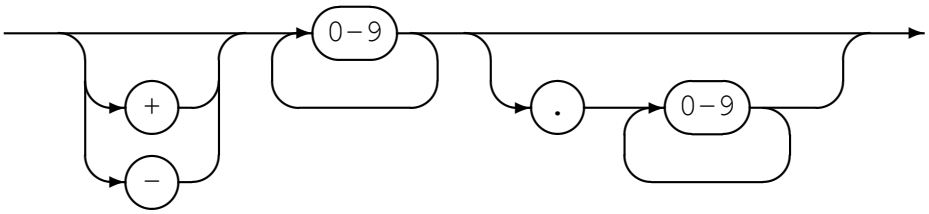
5.2.1 Lexical Elements

The base elements are *Literals* and *Identifiers*.

- Numeric Literals

  A Numeric literal is made up from an optional plus or minus sign followed by one or more digits and optionally followed by a point and one or more digits.
   ```

Number Literal



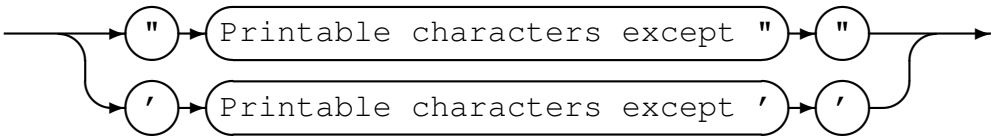
- String Literals

  String literals are made up from

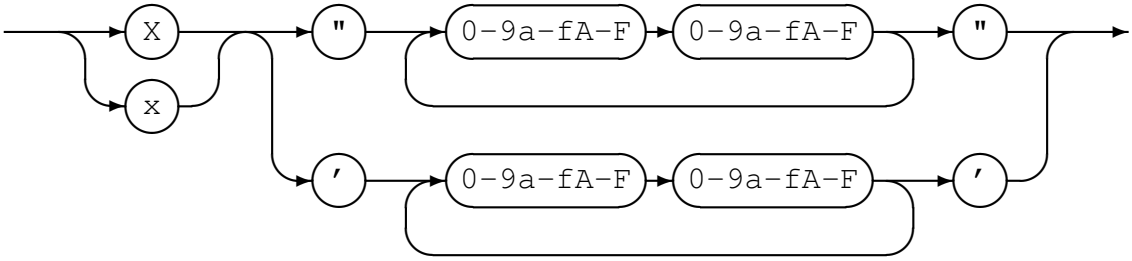
  – Any number of printable characters, except the enclosing character and a newline, enclosed in either single or double quotes.

  – An even number of hexadecimal digits enclosed in either single or double quotes and prefixed with a lower or upper case `X`.

  String Literal



  Hexadecimal Literal



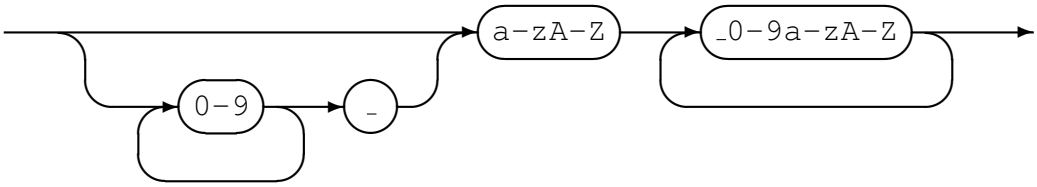
- Identifiers

  An identifier is used for both variable and function names. An identifier must conform to:

  – A lower or upper case alphabetic character followed by any number of underscores, decimal digits and upper and lower case alphabetic characters.

  – One or more decimal digits followed by an underscore and the above rule.

Identifier

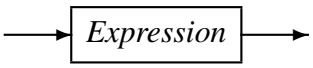


5.2.2 Syntactical Elements

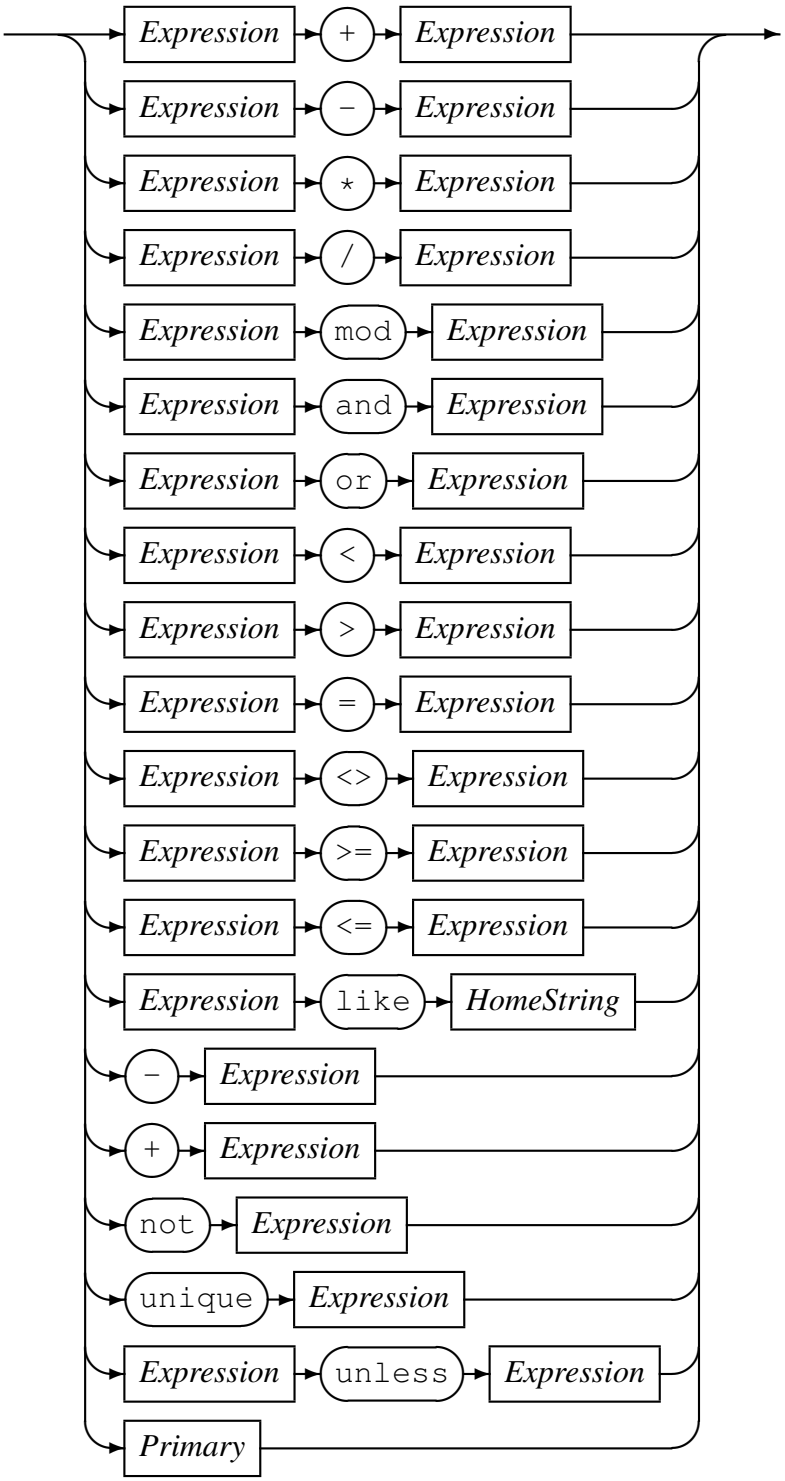
Expressions may themselves be used as syntactical elements when forming a compound expression.

The complete syntax of a compound expression is explained in the following sections starting with the compound expression and working down to the lowest level syntactic element.

CompoundExpression



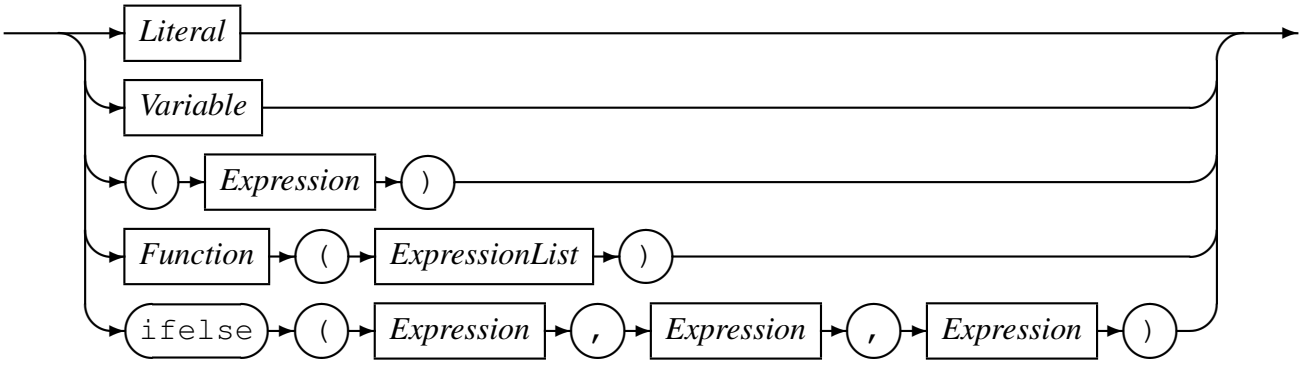
Expression



The `unless`-operator conditionally returns the value of the right-hand operand, unless there is an error evaluating the right-hand operand. In the case where the right-hand operand fails to evaluate to a proper value, the value of the left-hand operand is returned

instead. The left-hand operand is always evaluated before the right-hand operand. If the left-hand operand fails to evaluate to a proper value, then the result of the `unless`-operator is a failure.

Primary



As a terminal in the syntax structure an expression or *Primary* is either a *Literal* or a *Variable*, an *Expression* enclosed in parenthesis, a *Function* call reference, or the conditional evaluation operator `ifelse`. A *Literal* may be a *String Literal* or a *Number Literal* as described in Section 5.2.1 on page 34.

Where required by the encoding indicated or defaulted, characters representing the attribute value of a string are changed to an alternate character set if the required character set is not the same as the home character set being used. For example, on a machine in which the characters are naturally represented using the EBCDIC character set encoding (such as code page of 1047 or Latin 1/Open Systems), if the data being processed is from a machine in which the characters are naturally represented using the ASCII character set (such as ISO8859-1), then the characters in the String literal (assumed to be represented in EBCDIC) will be translated to their corresponding ASCII characters for processing. This does not apply to String literals that were represented as a sequence of hexadecimal digits.

Both a *Function* (see Section 5.2.2 on page 40) and an *Expression* are made up of sub-expressions, although eventually even they must terminate and resolve to a value.

A *HomeString* is a *String Literal* that may not be represented as a sequence of hexadecimal digits, but in which the encoding is left in the natural encoding of the machine processing the data; that is the machine on which the expression string is being compiled. This is required for the right-hand operand of the like operator as this operator translates the value of the left-hand operand into the local encoding when performing pattern matching.

Operators, variables and functions are described in more detail below:

- Operators

  In the context of the expression evaluation library, an operator is a symbol that

operates on or causes an action to be be performed on the constants and variables adjacent to it. An operator is either

- Monadic
  A monadic operator only operates on one value and usually employ either prefix or postfix notation in that they either occur before or after the value they operate on. The expression evaluation library uses only prefix monadic operators.
- Dyadic
  Dyadic operators operate on two values and employ infix notation in that they operate on the the values that immediately precede and follow the operator.

All operators return a value of a defined type which is the result of the computation. The type returned by an operator must be semantically consistent within the context of the rest of the expression and the grammar it may be embedded in.

Table 2 on page 39 lists the allowed operators, their precedence, associativity, arity (whether or not they are monadic or dyadic) and Type.

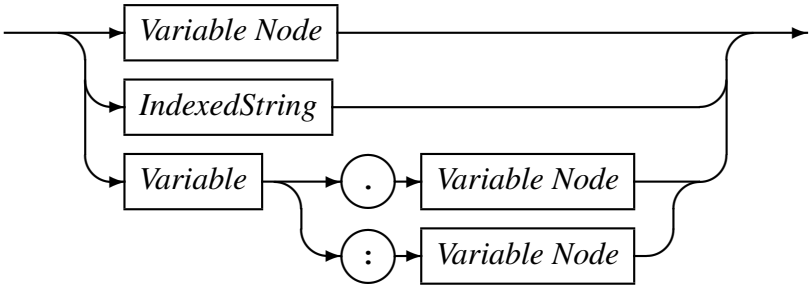
Operator	Precedence	Associativity	Arity	Type
`like`	1	non-assoc	dyadic	Relational
`<>`	1	left	dyadic	Relational
`>=`	1	left	dyadic	Relational
`<=`	1	left	dyadic	Relational
`=`	1	left	dyadic	Relational
`>`	1	left	dyadic	Relational
`<`	1	left	dyadic	Relational
`+`	2	left	dyadic	Arithmetic
`-`	2	left	dyadic	Arithmetic
`or`	2	left	dyadic	Boolean
`*`	3	left	dyadic	Arithmetic
`/`	3	left	dyadic	Arithmetic
`div`	3	left	dyadic	Arithmetic
`and`	3	left	dyadic	Boolean
`mod`	3	left	dyadic	Arithmetic
`-`	4	left	monadic	Arithmetic
`not`	4	left	monadic	Boolean
`unique`	4	left	monadic	boolean
`unless`	5	left	dyadic	boolean

Table 2: Operators: Precedence, Associativity, Arity and Type

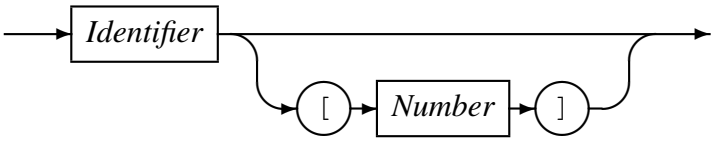
- Variables

A variable is the name of a storage location that holds a value. Simply this name is just an *Identifier*, but may be more than one level or node including an index.

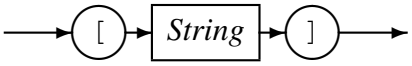
Variable



Variable Node



IndexedString



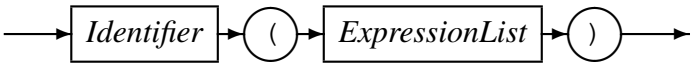
Examples of variable names are:

- `Address` - A single node variable with no indexing.
- `Customer.Address` - A two node variable.
- `Customer.Address[1]` - A two node variable where the Address portion of the variable is the first of an array of items. Here this may be the first line of an address.
- `Customer[3].Address[1]` - A two node variable that specifies the third entry of the Customer array and the first entry of the Address array within that Customer.
- `Customer.Contact.HomePhone` - A three node variable.

- Functions

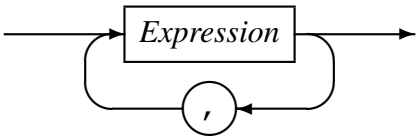
  A function is a special type of operator. It is specified by the function name, an identifier, followed by a comma separated list of arguments enclosed in parentheses.

  Function



where an expression list is defined as

ExpressionList



The function call is replaced with the result of the call and the result type must be semantically consistent within the context of the rest of the expression and the grammar it may be embedded in.

5.3 Built-in Functions

Functions for expression evaluation can be supplied by the application that uses it and as such has a rich set of plug in functions that can not be documented here. However there are functions that are common to all data processing and these are supplied by the expression evaluation library and are described below.

5.3.1 SysStrLen, strlen, length

- **Synopsis**
  - `SysStrLen(string)`
  - `strlen(string)`
  - `length(string)`
- **Parameters**
  - Parameter 1 type: String.
- **Description**

  The `SysStrLne` function (aliases `strlen`, `length`) returns the number of characters in the string supplied as the first argument.

5.3.2 SysSubStr, substr

- **Synopsis**
  - `SysSubStr(string,start,length)`
  - `substr(string,start,length)`
- **Parameters**

  - Parameter 1 type: String.
  - Parameter 2 type: Number.
  - Parameter 3 type: Number.
  - Return type: String.

- **Description**

  The `SysSubStr` function (alias `substr`) returns a substring of the given string from start for length characters or the remainder of string whichever is the shortest.

  The start must be greater than zero and the length must be zero or greater. If the start position is past the end of the string then a NULL string is returned.

5.3.3 SysString, string

- **Synopsis**
  - `SysString(number)`
  - `string(number)`
- **Parameters**
  - Parameter 1 type: Number.
  - Return type: String.
- **Description** The `SysString` function (alias `string`) returns the value of number as a string.

5.3.4 SysNumber, number

- **Synopsis**
  - `SysNumber(string)`
  - `number(string)`
- **Parameters**
  - Parameter 1 type: String.
  - Return type: Number.
- **Description** The `SysNumber` function (alias `number`) returns a number equivalent to the value of string.

5.3.5 SysStrCat, strcat

- **Synopsis**
  - `SysStrCat(first,second)`
  - `strcat(first,second)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** The `SysStrCat` function (alias `strcat`) returns a String which is the concatenation of the two input strings `first` and `second`.

5.3.6 SysStrStr, strstr

- **Synopsis**
  - `SysStrStr(haystack,needle)`
  - `strstr(haystack,needle)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** The `SysStrStr` function (alias `strstr`) returns the start position of `needle` within `haystack`. If `needle` does not occur in `haystack` then zero is returned, otherwise the position (origin 1) is returned.

5.3.7 SysStrSpn, strspn

- **Synopsis**
  - `SysStrSpn(string,accept)`
  - `strspn(string,accept)`
- **Parameters**
  - Parameter 1 type: String.

  - Parameter 2 type: String.
  - Return type: Number.

- **Description** The `SysStrSpn` function (alias `strspn`) returns the number of characters (bytes) in the initial segment of `string` which consist only of characters from `accept`.

5.3.8 SysStrCspn, strcspn

- **Synopsis**
  - `SysStrCspn(string,reject)`
  - `strcspn(string,reject)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** The `SysStrCspn` function (alias `strcspn`) returns the number of characters (bytes) in the initial segment of `string` which do not match any character from `reject`.

5.3.9 SysStrPadRight, padright

- **Synopsis**
  - `SysStrPadRight(string,length,pad)`
  - `padright(string,length,pad)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: Number.
  - Parameter 3 type: String. Although the type is String, only the first character is used as the pad character.
  - Return type: String.
- **Description** The `SysStrPadRight` function (alias `padright`) returns a string whose length is `length` and:

- if `length` is greater than the length of `string`, is `string` padded on the right with the `pad` character
- if `length` is less than the length of `string`, is `string` truncated from the right to `length`.
- if `length` is equal to the length of `string`, is `string`.

5.3.10 SysStrPadLeft, padleft

- **Synopsis**
  - `SysStrPadLeft(string,length,pad)`
  - `padleft(string,length,pad)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: Number.
  - Parameter 3 type: String. Although the type is String, only the first character is used as the pad character.
  - Return type: String.
- **Description** The `SysStrPadLeft` function (alias `padleft`) returns a string whose length is `length` and:
  - if `length` is greater than the length of `string`, is `string` padded on the left with the `pad` character
  - if `length` is less than the length of `string`, is `string` truncated from the left to `length`.
  - if `length` is equal to the length of `string`, is `string`.

5.3.11 SysFmtCurrTime, strftimecurr

- **Synopsis**
  - `SysFmtCurrTime(format)`
  - `strftimecurr(format)`
- **Parameters**
  - Parameter 1 type: String.
  - Return type: String.

- **Description** The `SysFmtCurrTime` function (alias `strftimecurr`) returns a string that represents the current time as formatted according to `format` using the C run-time `strftime()` function. Common values for `format` are:
  - %c - The preferred date and time representation for the current locale.
  - %d - The day of the month as a decimal number (range 01 to 31).
  - %F - Equivalent to %Y-%m-%d (the ISO 8601 date format).
  - %H - The hour as a decimal number using a 24-hour clock (range 00 to 23).
  - %j - The day of the year as a decimal number (range 001 to 366).
  - %m - The month as a decimal number (range 01 to 12).
  - %M - The minute as a decimal number (range 00 to 59).
  - %s - The number of seconds since the Epoch, 1970-01-01 00:00:00
  - %S - The second as a decimal number (range 00 to 60, allows for leap seconds).
  - %T - The time in 24-hour notation (%H:%M:%S).
  - %y - The year as a decimal number without a century (range 00 to 99).
  - %Y - The year as a decimal number including the century.
  - %% - A literal '%' character.
  - Any other characters, not specified by `strftime()`, are copied verbatim from `format` to the result string.

5.3.12 SysTime, time2epoch

- **Synopsis**
  - `SysTime(datetime,format)`
  - `time2epoch(datetime,format)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String. Default "%Y%m%d".
  - Return type: Number.
- **Description** The `SysTime` function (alias `time2epoch`) returns the number seconds since the Epoch calculated from `datetime` under the specification of `format`.

The seconds since the Epoch, when interpreted as an absolute time value, represents the number of seconds elapsed since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`.

`datetime` must be a string representation of a date and / or time and `format` must be a date format string that exactly describes `datetime` using the format characters as specified and used by the C function `strptime()`.

Common options for the `format` are:

- %% - The % character.
- %c - The date and time representation for the current locale.
- %C - The century number (0-99).
- %d or %e - The day of month (1-31).
- %H - The hour (0-23).
- %I - The hour on a 12-hour clock (1-12).
- %j - The day number in the year (1-366).
- %m - The month number (1-12).
- %M - The minute (0-59).
- %p - The locale's equivalent of AM or PM. (Note: there may be none.)
- %S - The second (0-60; 60 may occur for leap seconds; earlier also 61 was allowed).
- %T - Equivalent to %H:%M:%S.
- %x - The date, using the locale's date format.
- %X - The time, using the locale's time format.
- %y - The year within century (0-99). When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969-1999); values in the range 00-68 refer to years in the twenty-first century (2000-2068).
- %Y - The year, including century (for example, 1991).

5.3.13 SysStrFTime, strftime

- **Synopsis**
  - `SysStrFTime(seconds,format)`
  - `strftime(seconds,format)`

- **Parameters**
  - Parameter 1 type: Number.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** The `SysStrFTime` function (alias `strftime`) returns a string date time representation of `seconds` formatted according to `format` as described in the C runtime function `strftime()`.

  `seconds` is the number of seconds since the Epoch, which when interpreted as an absolute time value, represents the number of seconds elapsed since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`.

  `format` must be a date format string used to format the returned date time string. For common values of `format` see section 5.3.11 on page 46

5.3.14 SysInTable, intable

- **Synopsis**
  - `SysInTable(table,search)`
  - `intable(table,search)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Boolean.
- **Description**

  The `SysInTable` function (alias `intable`) returns a boolean `TRUE` if the value of `search` is found in the table of items `table`, otherwise it returns a boolean `FALSE`.

  The value of `table` may be either the name of a text file in which each line is one element of the table, or a comma (,) or semi-colon (;) delimited string of the element values of the table.
- **Examples**
  - SysInTable("C:\customerNames.txt","Smith") This will test whether the name "Smith" occurs in the list of elements in the file `C:\customerNames.txt`.

- SysInTable("/tmp/customerNames.txt",Record.Surname) This will test whether the name identified by the object types[6] field `Record.Surname` occurs in the list of elements in the file `/tmp/customerNames.txt`.
- SysInTable("Smith,Jones,Right",Record.Surname) This will test whether the name identified by the object types[6] field `Record.Surname` occurs in the list of elements in the comma separated list specified by the first argument.

5.3.15 SysStrCondPack, condpack

- **Synopsis**
  - `SysStrCondPack(String,String)`
  - `condpack(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description**

  The `SysStrCondPack` function (alias `condpack`) returns a string which is conditionally formed by packing the string passed in the first parameter using the second parameter as a possible replacement character. If the first parameter matches the regular expression `X"[0-9][A-F][a-f]"` then the hexadecimal characters are packed into the corresponding encoding character set (ASCII or EBCDIC) characters. If the second parameter does not have a zero length, then the first character of this parameter string is used to replace all the non-graphic/non-printable characters of the packed character string. When the second parameter string has a zero length, then the character "`?`" is used as the replacement character for non-graphic/non-printable characters in the return string.

  If the first parameter string does not match the regular expression then the string is considered to already be packed. In this case, the string is still checked if the second parameter length is greater than one and the non-graphic/non-printable characters are replaced by the first character of the second parameter string. When the second parameter string has a zero length, then the character "`?`" is used as the replacement character for non-graphic/non-printable characters in the return string.
- **Examples**

- `condpack('X"414141"',"?")` on an ASCII based machine returns the string `AAA`.
- `condpack('X"4141410000"',"?")` on an ASCII based machine returns the string `AAA??`.
- `condpack("4141410000","?")` on an ASCII or EBCDIC based machine returns the string `4141410000`.

5.3.16 TermAppStructDataGet, sfget

- **Synopsis**
  - `TermAppStructDataGet(String,String)`
  - `sfget(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** This function takes as the first parameter a value that should contain a `TermApp DE48-F0.16` Structured Data field and as the second parameter the name of a field within the structured data. The function will return the value of the named field as a string, if the name could not be found an empty string is returned.
- **Examples**
  - `sfget(DE48_FIELD,"OSVer")`
    Where**DE48_FIELD=**
    `219Postilion::MetaData275211FWSerialNbr11115SWRel111 19CommsType11118TermType11115OSVer11116SWHash111211F 01E201WSerialNbr2210100010000000100224231 5SWRel213144060219CommsType214INTERNAL MODEM 18TermType18EFTsma rt15OSVer19820036078 16SWHash18B4E1963A`
    returns the string `820036078`

5.3.17 TermAppStructDataSet, sfset

- **Synopsis**
  - `TermAppStructDataSet(String,String,String)`

- `sfset(String,String,String)`

- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Return type: String.
- **Description**
- **Examples**
  - ```
    sfset(DE48_FIELD,"FWSerialNbr",
          "+----------LongerValue----------+")
    ```
    Where **DE48_FIELD** is initially set to
    ```
    219Postilion::MetaData275211FWSerialNbr11115SWRel111
    19CommsType11118TermType11115OSVer11116SWHash111211F
    WSerialNbr2210100010000000100224231 5SWRel2131401E201
    4060219CommsType214INTERNAL MODEM18TermType18EFTsmar
    t15OSVer1982003607816SWHash18B4E1963A
    ```
    Will return the updated value of **DE48_FIELD** as
    ```
    219Postilion::MetaData275211FWSerialNbr11115SWRel111
    19CommsType11118TermType11115OSVer11116SWHash111211F
    WSerialNbr233+----------LongerValue----------+15SWRe
    l2131401E2014060219CommsType214INTERNAL MODEM18TermT
    ype18EFTsmart15OSVer1982003607816SWHash18B4E1963A
    ```

5.3.18 gsub, replace

- **Synopsis**
  - `gsub(String,String,String,String)`
  - `replace(String,String,String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Parameter 4 type: String.
  - Return type: String.
    ```

- **Description** The function gsub() operates in much the same why as the `awk gsub` function does. The four parameters are

  1. Regular Expression (r) This parameter is a regular expression that should match one or more portions of the input text (t).
  2. Substitution String (s) This parameter is the replacement string
  3. Text to operate on (t) This parameter is the original input text value.
  4. How to operate (h) This parameter determines how many times the replacement text is substituted.

  How (h) can be either

  – g or G which means replace all occurrences of matched text with the substitution string.

  – Numeric which means replace only that occurrence.

  The regular expression (r) matches none, one or more portions of the input text (t) and based on the value of how (h) gsub() returns the input string where one or all of the matches are replaced with the substitution string (s).

- **Examples**

  – `gsub("a","bb",textfield,how)` This example specifies to replace the letter `a` with two letter `b`'s in `textfield` under the control of the variable `how`.

Textfield value	How	Returned Value	Description
abcdea12345a	G	bbbcdebb12345bb	Each a is replaced by two b's.
abcdea12345a	2	abcdebb12345a	The second a is replaced by two b's.
abcdea12345a	1	bbbcdea12345a	The first a is replaced by two b's.

Table 3: Effect of using gsub() to substitute text

  – `gsub("\([^ ]\+\) \([^ ]\+\)","\2 \1",textfield,how)` This example specifies to match two substrings that contain any character except a space and that the first substring must be followed by a space followed by the second substring. The substitution string specifies to replace the whole matched value with the second matched substring followed by a space followed by the first matched substring. In other words is swaps two substrings around where the substrings do not contain a space and are separated by one space. The number of times the replacement is done is governed by the value of the variable `how`.

Textfield value	How	Returned Value	Description
ABC DEF	G	DEF ABC	The order of the two strings is reversed.
A1 bA1 A2 BA2	G	bA1 A1 BA2 A2	Each set of two strings are reversed.
A1 bA1 A2 BA2	2	A1 bA1 BA2 A2	Only the second set is reversed.

Table 4: Effect of using gsub() to substitute text

5.3.19 alias, lookup

- **Synopsis**
  - `alias(String,String)`
  - `lookup(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** This function uses the second parameter as a lookup key to extract the associated value in the first parameter, which holds keyword value pairs. The value corresponding to the matched key word is returned. The keyword value pairs specified in the first parameter can either be a comma or semi-colon list of `keyword=value` pairs or a file name containing one `keyword=value` pair per line.
- **Examples**
  - `lookup("A=Alsatian,L=Labrador,S=Spaniel","L")`
    Will return the string "Labrador"
  - `lookup("D:/lookup.txt","L")`
    will return the string "Labrador" if the file `D:/lookup.txt` holds the following:

    ```
    A=Alsatian
    L=Labrador
    S=Spaniel
    ```

5.3.20 pstore_set, psset

- **Synopsis**
  - `pstore_set(String,String,String)`

  - `psset(String,String,String)`

- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Return type: String.
- **Description** This function sets a value in a persistent store specified in parameter 1 using the variable name specified in parameter 2 and the value in parameter 3. If an error occurs, for example not being able to connect to the persistent store server, an error condition is returned.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.
- **Examples**
  - `pstore_set("www.codemagus.com:60069","ServerName","theCloud")` Will set and return the value of the variable `ServerName` to `theCloud` on the specified host.
  - `psset("www.codemagus.com:60069","ServerName","theCloud")` Will perform the same function as the example above.

5.3.21 pstore_get, psget

- **Synopsis**
  - `pstore_get(String,String)`
  - `psget(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** This function retrieves a value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. If the named variable is not found then an error condition is returned.

The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.

- **Examples**
  - `pstore_get("www.codemagus.com:60069","ServerName")`
    Will return the value of the variable `ServerName` from the specified host.
  - `psget("www.codemagus.com:60069","ServerName")`
    Will perform the same function as the example above.

5.3.22 pstore_get_cset, psget_cset

- **Synopsis**
  - `pstore_get_cset(String,String,String)`
  - `psget_cset(String,String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Return type: String.
- **Description** This function retrieves a value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. If the named variable is not found then it is created with the default value specified in parameter 3 and that value is returned.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.
- **Examples**
  - `pstore_get_cset("www.codemagus.com","ServerName","theNet")`
    Will return the value of the variable `ServerName` from the specified host (using the default port), but if it is not found will return and set it to `theNet`.
  - `psget_cset("www.codemagus.com","ServerName","theNet")`

Will perform the same function as the example above.

5.3.23 pstore_get_incr, psget_incr

- **Synopsis**
  - `pstore_get_incr(String,String)`
  - `psget_incr(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** This function retrieves a string representation of a numeric value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. The numeric string is returned as a number type and is subsequently incremented by 1 and saved back to the persistent store as a numeric string.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.
- **Examples**
  - `pstore_get_incr("www.codemagus.com:60069","Count")`
    If the value of `Count` on the persistent store is 3, then this function will return 3 and store 4 back on the persistent store. If the variable `Count` is not found an error condition is returned.
  - `psget_incr("www.codemagus.com:60069","Count")`
    Will perform the same function as the example above.

5.3.24 pstore_get_incr_cset, psget_incr_cset

- **Synopsis**
  - `pstore_get_incr_cset(String,String,Number)`
  - `psget_incr_cset(String,String,Number)`
- **Parameters**

  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: Number.
  - Return type: Number.

- **Description** This function retrieves a string representation of a numeric value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. The numeric string is returned as a number type and is subsequently incremented by 1 and saved back to the persistent store as a numeric string. If the named variable is not found on the persistent store then the default value specified in parameter 3 is returned and subsequently incremented and saved on the persistent store.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.

- **Examples**

  - `pstore_get_incr_cset("codemagus","Count",17)`
    If the value of `Count` on the persistent store is 3, then this function will return 3 and store 4 back on the persistent store. If the variable `Count` is not found then the value 17 is returned and 18 is saved to the persistent store as the value of `Count`.

  - `psget_incr_cset("codemagus","Count",17)`
    Will perform the same function as the example above.

References

[1] recio: Record Stream I/O Library Version 1. CML Document CML00001-01, Code Magus Limited, July 2008. PDF.

[2] binary: Fixed and Variable Length Record Stream Access Method Version 1. CML Document CML00005-01, Code Magus Limited, July 2008. PDF.

[3] dataset: Catalog Access Method Definitions Version 1. CML Document CML00013-01, Code Magus Limited, July 2008. PDF.

[4] directory: Directory Record Stream Access Method Version 1. CML Document CML00014-01, Code Magus Limited, July 2008. PDF.

[5] MVS: MVS Record Stream Access Method Version 1. CML Document CML00016-01, Code Magus Limited, July 2008. PDF.

[6] objtypes: Configuring for Object Recognition, Generation and Manipulation. CML Document CML00018-01, Code Magus Limited, July 2008. PDF.

[7] remote: Remote Record Stream Access Method Version 1. CML Document CML00022-01, Code Magus Limited, July 2008. PDF.

[8] standard: Standard Input And Output Using Recio Version 1. CML Document CML00030-01, Code Magus Limited, July 2008. PDF.

[9] text: File Access Method Using POSIX Streams Version 1. CML Document CML00031-01, Code Magus Limited, July 2008. PDF.

[10] image: DB2 Image Copy Reader Access Method Version 1. CML Document CML00036-01, Code Magus Limited, July 2008. PDF.

[11] edit: Recio Edit Access Method Version 1. CML Document CML00047-01, Code Magus Limited, June 2009. PDF.

[12] db2query: Recio DB2 Query Access Method Version 1. CML Document CML00050-01, Code Magus Limited, November 2009. PDF.

[13] db2dclgen: Recio DB2 DCL Generator (DCLGN) Access Method Version 1. CML Document CML00051-01, Code Magus Limited, November 2009. PDF.

[14] applparms: Application Parameters Library User Guide and Reference Version 1. CML Document CML00054-01, Code Magus Limited, January 2010. PDF.

[15] null: Null Access Method Using Recio Version 1. CML Document CML00055-01, Code Magus Limited, January 2009. PDF.

[16] source: Source Access Method Using Recio Version 1. CML Document CML00056-01, Code Magus Limited, January 2009. PDF.

[17] debugapi: Debug API User Guide and Reference Version 1. CML Document CML00060-01, Code Magus Limited, February 2010. PDF.