AppBuilder

By Magic Software Enterprises

Magic Software AppBuilder

Version 3.2

Rules Language Reference Guide

Corporate Headquarters
Magic Software Enterprises
5 Haplada Street,
Or Yehuda 60218, Israel
Tel +972 3 5389213
Fax +972 3 5389333

© 1992-2013 AppBuilder Solutions
All rights reserved.
Printed in the United States of America.
AppBuilder is a trademark of AppBuilder Solutions. All other product and company names mentioned herein are for identification purposes only and are the property of, and may be trademarks of, their respective owners.

Portions of this product may be covered by U.S. Patent Numbers 5,295,222 and 5,495,610 and various other non-U.S. patents.
The software supplied with this document is the property of AppBuilder Solutions and is furnished under a license agreement. Neither the software nor this document may be copied or transferred by any means, electronic or mechanical, except as provided in the licensing agreement.
AppBuilder Solutions has made every effort to ensure that the information contained in this document is accurate; however, there are no representations or warranties regarding this information, including warranties of merchantability or fitness for a particular purpose. AppBuilder Solutions assumes no responsibility for errors or omissions that may occur in this document. The information in this document is subject to change without prior notice and does not represent a commitment by AppBuilder Solutions or its representatives.

1. Rules Language Reference Guide . . . 2
1.1 Introduction to the Rules Language . . . 2
1.2 Data Types . . . 6
1.3 Data Items . . . 18
1.4 Expressions and Conditions . . . 30
1.5 Functions . . . 44
1.5.1 Numeric Conversion Functions . . . 44
1.5.2 Mathematical Functions . . . 48
1.5.3 Date, Time and Timestamp Functions . . . 53
1.5.4 Character String Functions . . . 66
1.5.5 Double-Byte Character Set Functions . . . 74
1.5.6 Error-Handling Functions . . . 75
1.5.7 Support Functions . . . 76
1.6 Declarations . . . 81
1.7 Procedures . . . 95
1.8 Control Statements . . . 100
1.9 Assignment Statements . . . 112
1.10 Condition Statements . . . 127
1.11 Transfer Statements . . . 133
1.12 Macros . . . 147
1.13 Platform Support and Target Language Specifics . . . 173
1.13.1 Specific Considerations for C . . . 173
1.13.2 Specific Considerations for Java . . . 176
1.13.3 Specific Considerations for CSharp . . . 212
1.13.4 Specific Considerations for ClassicCOBOL . . . 212
1.13.5 Specific Considerations for OpenCOBOL . . . 215
1.13.6 Specific Considerations for ClassicCOBOL and OpenCOBOL . . . 233
1.13.7 Restrictions on Features . . . 235
1.13.8 Supported Functions by Release and Target Language . . . 237
1.14 Code Generation Parameters and Settings . . . 241
1.15 Reserved Words . . . 262
1.16 Decimal Arithmetic Support . . . 269
1.17 Rules Language Quick Reference and Syntax . . . 278

Rules Language Reference Guide

This guide explains how to use the AppBuilder Rules Language to define the processing logic of an application. With the Rules Language, an application can transfer processing control from one rule to another, open windows at runtime, generate reports, pass data, and define how the entities comprising an application interact.

Rules Language is supported on C, Java, ClassicCOBOL, and OpenCOBOL language platforms. Some features are platform-specific. For more information about target language specifics, see Platform Support and Target Language Specifics.

This guide includes the following topics and sections:

- Introduction to the Rules Language
- Data Types
- Data Items
- Expressions and Conditions
- Functions
- Declarations
- Procedures
- Control Statements
- Transfer Statements
- Macros
- Platform Support and Target Language Specifics
- Code Generation Parameters and Settings
- Reserved Words
- Decimal Arithmetic Support
- Rules Language Quick Reference and Syntax

Introduction to the Rules Language

This guide explains how to use the AppBuilder Rules Language to define the processing logic of an application. With the Rules Language, an application can transfer processing control from one rule to another, open windows at runtime, generate reports, pass data, and define how the entities comprising an application interact.

Rules Language is supported on C, Java, ClassicCOBOL, and OpenCOBOL language platforms. Some features are platform-specific. For more information about target language specifics, see Platform Support and Target Language Specifics.

The following topics are included in this overview:

- Rules Language Elements
- Syntax Flow Diagrams Conventions and Symbols
- Rules Language Statements and Arguments
- Documentation Conventions and Symbols

Although the Rules Language is a high-level language and hides most of the low-level details, a basic knowledge of programming concepts is required for writing rules code. Additionally, knowledge of SQL is recommended for coding rules that access databases.

Rules Language Elements

AppBuilder application consists of the set of executable modules which define the application logic. The term Rule is used to denote the module of this kind. To define the data and process flow within the Rule as well as interaction between the different Rules a simple procedural language is used which is called Rules Language.

The following table describes the Rules Language statements:

Rules Language Statements

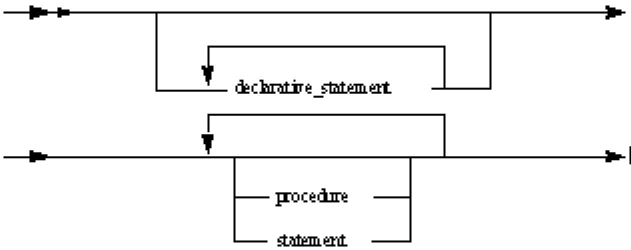
Statements	Description
Declarative statement (declaration)	The binding of an identifier to the information that relates to it. For example, to declare a variable means to address a portion of memory, bind it with the information about the data type of the variable, and label it with the variable name. This is also true when declaring a constant. The difference is that since a constant, by definition, always has the same value, it can be initialized and bound with its value at the time when it is declared.
Procedural statement (procedure)	A named sequence of statements, often with associated data types and variables, that usually performs a single task. Procedures may have parameters and can return value to their caller.

Comment statement	A description that explains aspects of a part of the program to other programmers, for example, and has no impact on the execution of the program.
Assignment statement	A statement that assigns a value to a variable. It is a good practice to assign a value to a variable prior to its usage; however, since all variables are initialized with valid values in the Rules Language, it is not obligatory to do so. These statements usually consist of three elements: an expression to be assigned, an assignment operator, and a destination variable.
Control statement	A statement that affects the flow of execution through a Rule. These include conditional statements, iterative statements, and transfer statements.
Transfer statement	A statement in a programming language that transfers the flow of execution to another location in the program. For example, a USE RULE statement.
File access statement	A SQL statement that performs database-related tasks such as fetching data from the database and updating data in the database.
Condition statement	A statement that alters the flow of execution depending on some condition. An example of condition statement is an IF statement.

This overview describes the elements of the Rules Language from the top level (statement) down to the lowest level (arguments); however, throughout this reference documentation, the Rules Language elements are listed from the lowest level up. For instance, data items are discussed before expressions, and expressions before statements. Thus, the documentation describes Rules Language building blocks first and then the statements used to combine these blocks into rules.

The following figure shows an example of a simple rule syntax diagram with a declaration, procedure, and statement.

Rule diagram example



Syntax Flow Diagrams Conventions and Symbols

Grammar

This specification presents syntax of the Rules Language.

The grammar consists of a number of syntactic productions. Each production defines a notion of the Rules Language.

Each grammar production contains of a left-side of the production, followed by a colon symbol, followed by a right-side of the production. Each left-side of the production is a non-terminal symbol being defined by this production. Each right-side of the production is a sequence of non-terminal or terminal symbols.

Example.

date_data_type:

DATE

In this production a date data type is defined as the Rules Language keyword DATE. The left-side of the production is the non-terminal date_data_type, and the right-side of the production is the terminal symbol that is the Rules Language keyword.

Sometimes the non-terminal symbol is defined as a set of different cases.

For example, the Rules Language defines an assign statement that is either an map statement or a set symbol. It is possible to define the assign statement in the following manner:

assign_statement:

map_statement

assign_statement:

    set_statement

On other hand it is possible to define assign statement using only one right-side:

assign_statement:

    map_statement

    set_statement

In grammar productions, terminal symbols are shown as words in lower-case letters.

The terminal symbols are complied with the following conventions:

- Rules Language keywords are represented as a word in all CAPITAL LETTERS.
- Other terminal symbols are shown as words in *italics*. Such terminal symbols are data item that you provide when coding the statement, such as a name of a field or a string literal.

Besides of terminal symbols and non-terminal symbols in the right-side of the grammar production can be used the metacharacters. If the terminal symbol represented by single character matches the metacharacter then such terminal symbol is enclosed in apostrophes. The full list of the metacharacters is [,], (,), *.

- The parenthesis (,) can be used in the right-hand part of the grammar production to group a sequence composed of terminal and/or non-terminal symbols.
- The square brackets are used to indicate an optional symbol.
- The metacharacter * following an element (i.e. a terminal symbol or a non-terminal symbol or a sequence of terminal/non-terminal symbols enclosed in parentheses) indicates repetition of the symbol or the sequence zero or more times.

Example 1. The following production defines syntax of statement list

statement-list:

    statement*

This means that the statement list can be empty or contain a number of the statements.

Example 2. The production

IF-statement:

    IF condition statement-list [**ELSE** statement-list] **ENDIF**

is shorthand for:

IF-statement:

    IF condition statement-list **ENDIF**

    IF condition statement-list **ELSE** statement-list **ENDIF**

and defines an *IF-statement* to consist of the token **IF**, followed by a *condition*, followed by a *statement-list*, followed by optional **ELSE** *statement-list*, followed by the token **ENDIF**.

The different cases are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

date_time_data_type: one of

    DATE TIME TIMESTAMP

is shorthand for:

date_time_data_type:

    DATE

    TIME

TIMESTAMP

Case Sensitivity

Rules Language is *not* case-sensitive. Capital letters are used in the syntax diagrams only to indicate Rules Language keywords. Rules Language *does* consider case in character literals and macro names.

Reading a Diagram

Follow these steps to interpret the syntax of a diagram:

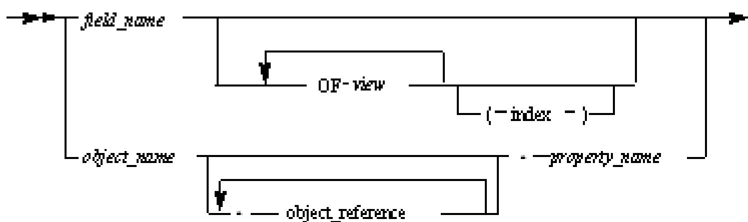
1. Start at the double-headed arrow on the left side and proceed to the right until you reach the end of the diagram.
2. Follow any one of the possible line paths from left to right. Any path that you can traverse from left to right results in valid syntax. For whichever line you follow, you must use all the words or symbols designated on that line.
3. You cannot go back to the left unless there is a loop. A loop is indicated by an arrow with its arrow head on its left end and appears above another line. You can repeat a loop any number of times.

Symbols used in syntax flow diagrams

Symbol	Meaning
	Flow of statement starts
	Flow continues on next line or may include another path
	Flow continued from previous line
	Flow may branch in either direction
	Flow of statement ends

For example, all of the following are ways a variable data item can appear according to the following diagram:

- *field_name*
- *field_name* OF view(5)
- *field_name* OF view OF view
- view.*field_name*



Rules Language Statements and Arguments

Each Rules Language statement consists of one or more clauses. A clause is a Rules Language keyword followed, in most cases, by one or more arguments. The actual arguments for each clause vary by statement. However, an argument can always be described as one of the following:

Data Item	Either a constant (a numeric value, a character value, a boolean value, or a symbol) or a variable (a view or a field).
Expression	Data items, other expressions, and functions (such as ROUND or STRLEN) linked with operators (such as + or -).
Condition	Expressions linked with relational operators (such as = or >) and other conditions linked with boolean operators (such as OR, AND, and NOT).
SQL statement	An argument only for an SQL ASIS statement.
other	Another statement or the name of a specific entity (such as a window or report).

Documentation Conventions and Symbols

When describing a statement within text, a keyword appears in all CAPITAL letters. In addition, its arguments are generally replaced by ellipses or dropped if they trail the statement. For example:

```
IF condition statement ELSE statement ENDIF
```

appears in text as:

```
IF...ELSE...ENDIF
```

while

```
CLEAR variable_data_item
```

appears simply as:

```
CLEAR
```

Data Types

A Rules Language data item must be defined as a specific data type. The data type of a data item determines both its compatibility with other data items and how the data item is stored or displayed. You declare a data item as having a certain data type either locally using the DCL statement or as a property of a field. See Declarations for more information.

See Data Type Conversions for information about converting a data item of one data type to a different data type.

Data Types

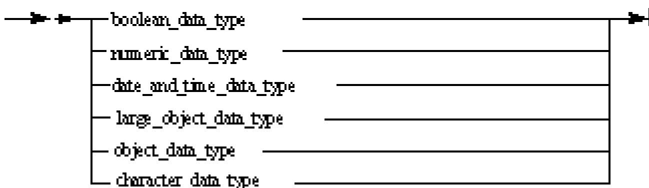
The data types used in the Rules Language are categorized as follows:

- BOOLEAN Data Type
- Numeric Data Types
- Date and Time Data Types
- Large Object Data Types
- Object Data Types
- Character Data Types

For platform specific information about data types, see the following:

- Data Types in C
- Data Types in Java
- Data Types in ClassicCOBOL
- Data Types in OpenCOBOL

Data Type Syntax



BOOLEAN Data Type

The BOOLEAN data type can have either of two values: TRUE or FALSE, which are reserved words by default. Boolean variables can be used anywhere in place of a condition in a rule, and results of conditional expressions can be mapped to these variables.

Boolean Syntax

Example: Using the BOOLEAN Data Type

The following routine uses a BOOLEAN data type to control processing flow.

```
DCL
   b BOOLEAN;
   i, j INTEGER;
ENDDCL

MAP 1 TO i
MAP 2 TO j
MAP i > j TO b  *> False <*
MAP TRUE TO b
IF b
   MAP 10 TO i  *> This line is executed. <*
ELSE
   MAP 1 TO i   *> This line is not. <*
ENDIF
```

Numeric Data Types

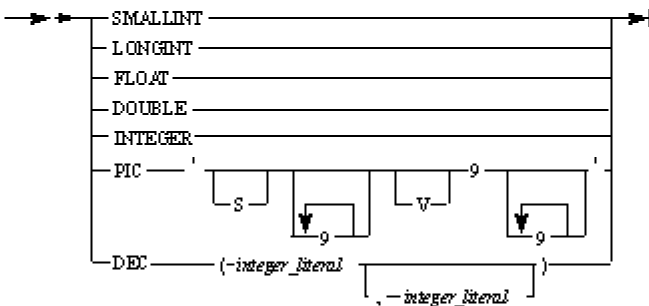
There are four numeric data types:

- SMALLINT
- INTEGER
- PIC
- DEC
- LONGINT, FLOAT, DOUBLE – See LONGINT, FLOAT and DOUBLE in Java for more details about these data types.

A data item of these types:

- can contain only numeric characters.
- can be preceded by a negative sign, indicating a negative value. The absence of a sign indicates a positive value (preceding a value with a positive sign is not valid).
- can have only *one* decimal point if it is a PIC or DEC data type.
- cannot have *any* decimal point if it is an INTEGER or SMALLINT data type.

Numeric Data Syntax



where:

- *integer_literal* is an integer value specifying the total length of the data item and the scale.

Locally-declared Numeric Data

See Local Variable Declaration for more examples of locally declared data items that use a numeric data type.

Consideration for COBOL

For ClassicCOBOL specifics including decimal field representation and DDL, see Decimal Field Representation in ClassicCOBOL.

For OpenCOBOL considerations regarding DDL, see DEC in OpenCOBOL.

SMALLINT

Use SMALLINT for a two-byte integer data item that contains values between -32,768 and 32,767 inclusive.

The following example illustrates how to locally declare a variable as the SMALLINT data type:

```
DCL
   COUNTER_1,COUNTER_2 SMALLINT;
ENDDCL
```

INTEGER

Use INTEGER for a four-byte integer data item that contains values between -2,147,483,648 and 2,147,483,647 inclusive.

The following example illustrates how to locally declare a variable as the INTEGER data type:

```
DCL
   SUBTOTAL INTEGER;
ENDDCL
```

PIC

Declaring a data item as an integer picture (PIC) creates a storage picture that structures numeric data according to the following format.

PIC Data Item Codes

Code	Meaning
S	Signed number
9	Number placeholder
V	Decimal placeholder

For example, a PIC data item declared with the storage picture S999V99 can contain numeric data from -999.99 to 999.99.

A PIC declaration represents an internal or storage PICTURE and should not be confused with an external, display, or edit PICTURE. Also, you cannot write S9(3)V9(2) or S(3)9V(2)9 as in COBOL or PL/I.

Besides the syntax shown in the flow diagram, the following restrictions apply to a PIC string:

- It cannot contain more than thirty-one 9s
- It cannot contain embedded spaces

You can also declare a *picture with trailing sign* in Java, Open COBOL, and C#. For details about this picture type declaration, see PIC with trailing sign.

The following are advantages of using the PIC data items:

- On the host, a decimal (DEC) data item is stored packed two bytes to one; a PIC type is not.
- A PIC data item is the only numeric data type that can be used in a comparison to a character data type. An unsigned PIC can be compared to the following:
  - Any signed/unsigned numeric data type
  - CHAR
  - VARCHAR
  - TEXT
  - IMAGE
- An unsigned PIC can be assigned to the following types of fields:
  - Any signed/unsigned numeric data type

- CHAR
    - VARCHAR
    - TEXT
    - IMAGE
- An unsigned PIC can be concatenated with the following field types:
    - An unsigned PIC
    - CHAR
    - VARCHAR
    - TEXT
    - IMAGE
    - MIXED
    - DBCS

For releases that support DBCS, an unsigned integer picture can be assigned to a MIXED field. Also, an unsigned integer picture can be concatenated with DBCS and MIXED. See Restrictions on Features for more information.

DEC

Use DEC to specify a decimal data item. The first integer value after a DEC keyword is the total length of the data item; the second integer value is the scale, indicating the number of places to the right of the decimal point. If no scale value is specified, it is assumed to be 0, indicating an integer value. A DEC data item is always assumed to be a signed value.

The following restrictions apply to length and scale:

- Length must be greater than or equal to 1 (one) and less than or equal to 31 (thirty-one) (1 length 31)
- Length includes the scale, but not the decimal point.
- Scale, if specified, must be greater than or equal to 0 (zero), and less than or equal to length (0 scale length)

Date and Time Data Types

A data item declared as one of the following three data types must contain numeric data:

- DATE
- TIME
- TIMESTAMP

The format standards used for date and time data types are set during installation. The specific standard designated for your system is contained in a configuration file. The format for your system must match the format your database uses or errors occur. For instance, the delimiter designated in your language configuration file must match the delimiter used by your database or your rule will not prepare.

Date and Time Syntax

For OpenCOBOL considerations regarding Date and Time Syntax, see Date and Time Functions in OpenCOBOL.

Date and Time Formats

AppBuilder supports the standard date and time formats shown in the following table:

Standard Date and Time Formats

Standard	Date Format	Example	Time Format	Example
ISO (International Organization for Standardization)	yyyy-mm-dd	1996-11-30	hh.mm.ss	14.15.05
USA	mm/dd/yyyy	11/30/1996	hh:mm AM or PM	2:15 PM
EUR (European)	dd.mm.yyyy	30.11.1996	hh.mm.ss	14.15.05
JIS (Japanese Industrial Standard Christian Era)	yyyy-mm-dd	1996-11-30	hh:mm:ss	14:15:05
LOCAL (site defined)	Any site-defined form		Any site-defined form	

Converting Dates and Times

Use the date and time conversion functions discussed in Date, Time and Timestamp Functions to convert a variable from one date and time data type to another (or to an integer or character data type).

Locally-declared Date and Time

See Local Variable Declaration for examples of locally-declared data items that use the date and time data types.

DATE

Use DATE for a date data item. The value in the data item is the number of days past the date of origin. January 1, 0000 is the date of origin and has a date number of 1. A DATE variable has a length of four-bytes except for OpenCOBOL. For OpenCOBOL considerations regarding DATE, see DATE, TIME and TIMESTAMP in OpenCOBOL.

TIME

Use TIME for a time data item. The value in the data item is the number of milliseconds past midnight. The TIME data type has a length of four-bytes except for OpenCOBOL. For OpenCOBOL considerations regarding TIME, see DATE, TIME and TIMESTAMP in OpenCOBOL.

TIMESTAMP

Use TIMESTAMP for a time data item where you need greater precision than milliseconds. The TIMESTAMP data type has a length of 12 bytes except for OpenCOBOL. The TIMESTAMP data type consists of three independent subfields:

```
<DATE>:<TIME>:<FRACTION>
```

The <DATE> and <TIME> fields are for the DATE and TIME data types. <FRACTION> is platform-dependent and provides a more precise time measurement than the <TIME> field. The value in the FRACTION field is usually displayed in picosecond units, but the actual units used are system-dependent and are determined by the limitations of the operating system.

Although you can set the values for TIME and TIMESTAMP to contain seconds and milliseconds, some DBMSs might not support the time and timestamp values that AppBuilder allows.

For OpenCOBOL considerations regarding TIMESTAMP, see DATE, TIME and TIMESTAMP in OpenCOBOL.

Large Object Data Types

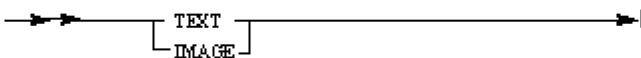
Use the following data type items to store a reference to a file containing a large object:

- TEXT
- IMAGE

On the workstation, a reference is a fully-qualified path and file name of a large-object file. On the host, a reference is a generated name, either generated automatically by AppBuilder when a large-object file is transferred to the host, or generated explicitly by the HPS_BLOB_GENNAME_FILE system component.

AppBuilder processes TEXT and IMAGE data items as if they were CHAR (256) data items, so any conditions that apply to CHAR data items also apply to TEXT and IMAGE data items. However, this does not apply to the files referenced by these data items.

Large Object Syntax



Mapping Character Values to TEXT or IMAGE

Use a MAP operation to assign a character value to a TEXT or IMAGE data item. A TEXT or IMAGE data item can also be mapped to another TEXT or IMAGE data item, or to a character field.

When a TEXT or IMAGE data item is mapped to another TEXT or IMAGE data item, only the large-object file name is copied. The large-object file itself is not copied.

Transferring a Large-Object File from the Workstation to the Host

Observe the following considerations when transferring a large-object file from a workstation to a host:

- In the workstation rule, map the full path and file name of the large-object file to a TEXT or IMAGE field in the input view of a host rule.
- When the workstation rule uses the host rule, AppBuilder automatically transfers the large-object file to the host. AppBuilder overwrites the TEXT or IMAGE field with the name of the large-object file on the host. If you want to later transfer the same large-object file, then copy the name from the TEXT or IMAGE field of the input view of the host rule to a TEXT or IMAGE field in the output view of the host rule. This makes the name available to the workstation rule — which can then pass it back to a host rule and request that the rule transfer the file.
- When a workstation rule uses a host rule, if the host rule maps the name of a large-object file to a TEXT or IMAGE field in its output view, AppBuilder automatically transfers the file to the workstation when the host rule returns control to the workstation rule.
- The recipient of a transferred file, whether a host or a workstation rule, is responsible for deleting the file when it is no longer required. AppBuilder will not delete the file because AppBuilder can not know when the file no longer needed.

TEXT

Use TEXT for a data item that holds a reference to a large-object, text file.

IMAGE

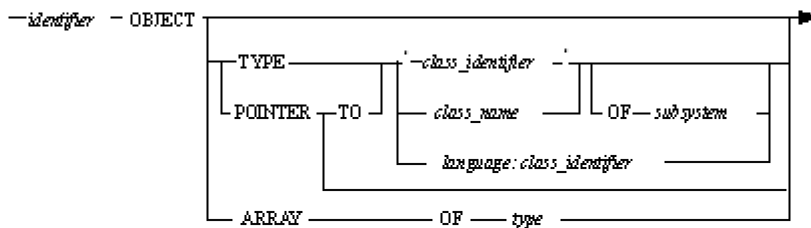
Use IMAGE for a data item that holds a reference to a binary large-object file (BLOB).

Object Data Types

The following are Object data types:

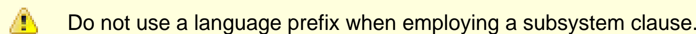
- OBJECT
- OBJECT POINTER
- Array Object

Object Data Type Declarations



where:

- *identifier* – see restrictions on names in Local Variable Declaration.
- *class_identifier* is a string that identifies the implementation of the class. Therefore, it might be the full Java class name for Java classes. The identification string is case-sensitive.
- *class_name* is a class name to be used in a rule. *Not* case-sensitive.
- *language* is a string that identifies the source of the class. The following languages are supported:
  - Java: the set of Java classes, available from CLASSPATH.



Do not use a language prefix when employing a subsystem clause.

- *subsystem* is the group to which this object belongs.

  The following subsystems are supported:
  - GUI_KERNEL: the set of AppBuilder-supplied window controls
  - JAVABEANS: used for any Java class

- *type* can be numeric, character, date and time, boolean, or object (with certain limitations, see Array Object for more details).

Refer to the *ObjectSpeak Reference Guide* for more information about using objects and AppBuilder-supplied objects.

Case-sensitivity in Identifiers

Generally speaking, any identifier without single quotation marks is *not* case-sensitive; likewise, any identifier with single quotation marks *is* case-sensitive. The only exception is the listener name in the LISTENER clause in an event or error handler declaration. See Event Procedure Declaration for the syntax used in this declaration.

OBJECT

The OBJECT data type is supported for all generations. However for C, ClassicCOBOL and OpenCOBOL only the first, simple form of the declaration is supported, i.e. the form: <identifier> OBJECT;

In C, OpenCOBOL and ClassicCOBOL generations OBJECT data type is equivalent to CHAR(8) and represents data object location. It can only be used in the following operations:

- mappings from the LOC function result. For details about LOC function see LOC.
- comparisons with other objects, as shown below:

```
dcl
   o1, o2 object;
   i integer;
enddcl

map LOC(i) to o1
map LOC(i) to o2

if o1 = o2
   trace("ok")
endif
```

In Java and CSharp the OBJECT data type is equivalent to the OBJECT POINTER data type. This data type represents a non-typed reference to an object. Since any object (object of any class) can be mapped to the OBJECT data type, it is useful when performing a type conversion. For more details see OBJECT and OBJECT POINTER in Java.

OBJECT POINTER

The OBJECT POINTER data type is supported only for generation to Java and CSharp; it is deprecated for C.

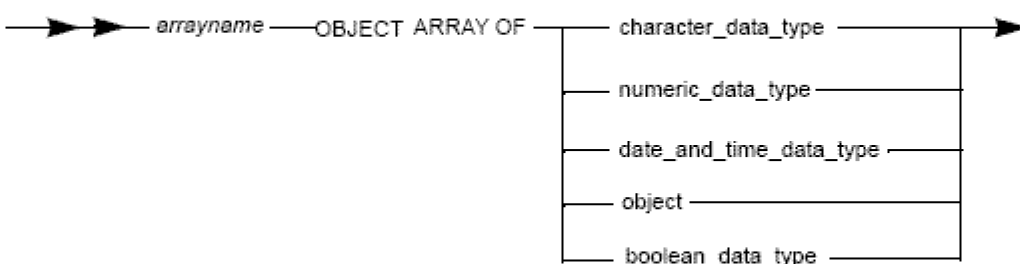
In Java and CSharp, the OBJECT POINTER data type is equivalent to the OBJECT data type.

Refer to OBJECT and OBJECT POINTER in Java for more details, CSharp generation supports the same Rules syntax and semantic as Java generation.

Array Object

Use the array object (OBJECT ARRAY form) to declare an array as a locally-declared data item.

OBJECT ARRAY Syntax



where:

- character_data_type — see Character Data Types.
- date_and_time_data_type — see Date and Time Data Types.
- numeric_data_type — see Numeric Data Types.
- object — see OBJECT. You can only have an array of non-typed objects, that is OBJECT ARRAY OF OBJECT.
- boolean_data_type — see BOOLEAN Data Type.

An array reference operation takes the form:

```
array_name.method(index)
```

Where *array_name* is the overall name of the array, the *method* following the delimiting period specifies a particular operation, and the value resulting from the evaluation of *index* specifies a particular member of the array.

For example:

```
DCL
   array1 OBJECT ARRAY OF INTEGER;
   array2 OBJECT ARRAY OF CHAR(20);
ENDDCL
```

Array Methods

The following methods can be applied to arrays:

- Append
- Size
- Elem
- Insert
- Delete

Append

This method appends one element at the end of an array. Its index is equal to the size of the array. This method takes one argument that must be of the same type as the array's type, or that can be converted to this type as if the argument were MAPped to a variable of array type. However, no warnings are issued if an argument cannot fit into the array type. See Data Type Conversions for compatible data types.

Size

This method takes no arguments and returns the size of the array. When first declared, an array has a size of zero. The size of an array is determined dynamically by how many elements you append to the array.

Elem

This method can have one or two arguments. If it is used to get the value of an array element, then it has one argument — the index of an existing element. For example:

```
MAP array1.elem(i+2) TO dec_value
```

After this statement is executed, field *dec_value* contains the same value as the array element with index i+2.

If this method is used as a destination in a MAP statement, it also must have one parameter — the index of an existing element. For example:

```
MAP char_value TO array2.elem(123)
```

After this statement is executed, the array element with index 123 contains the same value as the field *char_value*.

If the elem method has two arguments, then the first argument must be the index, and the second argument's value is assigned to the array element with the specified index.

For example:

```
array2.elem(123, char_value)
```

This statement has the same effect as the previous MAP statement; that is, the array element with index 123 will contain the same value as the field *char_value*.

In all cases, the index value must be within the range from 1 to the size of the array. Otherwise, a runtime error occurs.

Insert

This method has two arguments: the index and a value. A new array element containing the specified value is inserted into the array at the location specified by the index; the existing element at that location and all following elements have their index incremented by 1.

In all cases, the index value must be within the range from 1 to the size of the array. Otherwise, a runtime error occurs.

Delete

This function has only one argument, the index of an existing element. After the specified element is deleted, all element indices following the deleted element are decremented by 1. The following example deletes all elements from an array:

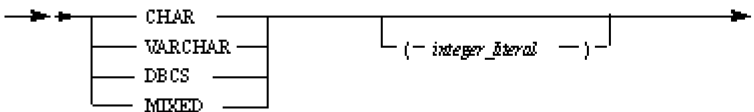
```
MAP arr.Size TO ArraySize
DO TO ArraySize
   arr.Delete(1)  *> every time the first element is deleted <*
ENDDO
```

Character Data Types

Character data types hold character data. The following are character data types in the Rules Language:

- CHAR
- VARCHAR
- DBCS and MIXED Data Types

Character Data Syntax



Locally-declared Character Data Items

See Local Variable Declaration for examples of locally-declared data items that use character data types.

Character Data Type Definitions

The descriptions for the various character data types use the terms "single-byte" and "double-byte" characters. These character types are defined as follows:

Single-byte

Single-byte characters occupy one byte in the current default or specified codepage. Field sizes are specified in characters but are allocated on the underlying assumption that one character equals one byte. Within a Unicode environment such as Java, while this single-byte concept is essentially meaningless, it still applies for data entering or exiting such an environment; for example, when making a remote rule call to a mainframe. The single-bye space character is the standard ASCII space character (or Unicode \u0020).

Double-byte

Double-byte character set (DBCS) uses 16-bit (two-byte) characters rather than 8-bit (one-byte) characters. Using double-byte characters expands the possible number of combinations of binary digits (1s and 0s) from 256 (as in ASCII) to 65,536 (or 256 x 256). Double-byte character sets are needed for such languages as Japanese, Chinese, and Korean, which have many characters.

Double-byte characters occupy two bytes in the current default or a specified codepage. Field sizes are specified in characters but are allocated

on the underlying assumption that one character equals two bytes, except in environments that have a specific type for such data such as a mainframe *graphic* type. Also, within a Unicode environment like Java, while there is no distinction between single-byte and double-byte characters, it is still relevant for data entering or exiting such an environment; for example, when making a remote rule call to a mainframe.

The double-byte space character is the ideographic or wide space Unicode character \u3000.

CHAR

Use CHAR for a fixed-length character data item. A fixed-length field reserves and retains the number of bytes you define in the field's length property. CHAR (*n*) denotes a CHAR data item of length *n*; a data item has a length of 1 if *n* value is not supplied. A CHAR data item is always padded with spaces to its declared length. The length of a CHAR data item is calculated in characters (or bytes) and can have a maximum length of 32K.

CHAR fields contain single-byte character data. No specific validation is performed on the content to ensure this. Rules Language statements will be validated to the extent that when mapping MIXED or DBCS fields or literals to a CHAR field, a conversion function must be explicitly specified. The only exception is a DBCS literal, which can be mapped directly to a CHAR field. When such data is in a CHAR field, it loses any special behavior or validation that might have been previously applied to it, thus it essentially becomes single-byte data, including being padded with single-byte spaces.

Trimming a CHAR field removes any trailing single-byte spaces. However, storing the trimmed result into another CHAR, MIXED or DBCS field will re-pad the data to the declared length of that field as necessary.

VARCHAR

Use VARCHAR for a variable-length character data item. VARCHAR (*n*) denotes a VARCHAR data item of maximum length *n*; a data item has a length of 1 if *n* value is not supplied. The length of a VARCHAR data item is calculated in characters or bytes. The maximum length is 32K.

Although the contents of a VARCHAR are variable length, the data item can be allocated based on the specified maximum length, therefore, do not assume that VARCHAR data items occupy less space. See Variable for the Length of the VARCHAR Data Item in ClassicCOBOL for information about ClassicCOBOL's use of VARCHAR in determining the length of a data item.

VARCHAR fields contain single-byte character data. No specific validation is performed on the content to ensure this. Rules Language statements will be validated to the extent that when mapping MIXED or DBCS fields or literals to a VARCHAR field, a conversion function must be explicitly specified. The only exception is a DBCS literal that can be mapped directly to a VARCHAR field. Once such data is in a VARCHAR field, it loses any special behavior or validation that may have been previously applied to it, thus it essentially becomes single-byte data. No padding beyond that already present in the source data will be added to the value.

Variable for the Length of the VARCHAR Data Item

Any variable of type VARCHAR implicitly declares a variable of type SMALLINT named:

<*varchar variable_name*>_LEN

For example, a variable named *VC* of type VARCHAR declares the variable:

VC_LEN of type SMALLINT

The *xxx_LEN* variable is a dynamic variable that represents the length of the VARCHAR variable. A dynamic variable can be changed directly through an assignment statement and these changes are reflected in the VARCHAR contents. At any given time, this field contains the actual length of the corresponding VARCHAR variable unless it has been changed directly.

However, the semantics of *xxx_LEN* (the way a change to *xxx_LEN* affects the corresponding VARCHAR data) varies on different platforms. Complete descriptions of the variations by platform are provided in the following sections:

- Variable for the Length of the VARCHAR Data Item in C
- Variable for the Length of the VARCHAR Data Item in Java
- Variable for the Length of the VARCHAR Data Item in ClassicCOBOL
- Variable for the Length of the VARCHAR Data Item in OpenCOBOL

Appbuilder 3.2 allows user to have explicitly defined xxx_LEN variable of type SMALLINT. Let's see an example:

```
DCL
   VC VARCHAR(10);
   VC_LEN SMALLINT;
ENDDCL
```

In this case all the references to *VC_LEN* it the rule text will be resolved to an explicitly defined variable rather than to internally defined one, so the assignment

```
MAP 22 TO VC_LEN
```

will not affect neither *VC* contents nor its length. Note that the warning will be generated in this case.

DBCS and MIXED Data Types

You can use a CHAR and VARCHAR data type only with single-byte character set (SBCS) data (that is, where each character is encoded as a single byte, allowing up to 256 distinct characters). However, some languages use character sets based on encoding characters using multiple bytes because they have more than 256 distinct characters. For those languages, AppBuilder contains two double-byte character set data types: DBCS and MIXED. See the chapter on DBCS Programing in the *Developing Applications Guide* for more details.

Only the DBCS enabled versions of AppBuilder support the DBCS and MIXED data types. Using these data types in other versions of AppBuilder causes the code generation step of the preparation process to fail.

See the following for more information about DBCS and MIXED data types:

- DBCS
- MIXED

DBCS

The DBCS data type can contain only fixed-length, double-byte character set data items. The length of a data item is defined as a number of double-byte characters, with a maximum of 32,767 characters (64k bytes).

Because field size is defined in terms of double-byte characters, the actual length is twice that number of bytes.

The concept of double-byte is dependent on the codepage. Even with codepages for the same language, a character might be double-byte on one platform but not the other. Depending on target platform, validation can optionally be performed at runtime to ensure only characters valid within the context of a specific codepage are allowed within a DBCS data item. It is assumed that the specified codepage will generally be that of the eventual destination platform for such data. It is also expected that the specified codepage might have to be a compromise because we must treat all DBCS fields similarly. Applying different validation rules for specific fields because they are not sent to the primary backend system is not a realistic option.

User input fields are validated for length to ensure no truncation of non-space data occurs when undergoing codepage conversion, such as when making remote rule calls. Such length validation must occur in the context of a specified codepage, and should typically be the one that is used by JNetE when marshalling data for transmission to backend systems. This is done in order to handle conversions from Unicode to other codepages, and stateful codepages for DBCS regions where characters or escape sequences are embedded in the data to switch modes, thus expanding the data.

Depending on target platform, when mapping to DBCS fields, the source data can be optionally validated for content to verify that all characters are double-byte, unless the source is another DBCS field. Additionally, the source data will be truncated if it is too long to fit in the destination field. Any such truncation is based on the actual field lengths, and will not take into account any length changes that might occur due to codepage conversion, such as when making remote rule calls. See the chapter on DBCS Programming in the *Developing Applications Guide* for more details.

A DBCS field will be padded to its declared length with double-byte spaces. Trimming DBCS data will remove any trailing double-byte spaces.

MIXED

The MIXED data type can contain double-byte characters and single-byte characters in any combination thereof. Although it might contain

double-byte characters, the length of a MIXED data item is declared on the basis of single-byte characters. And thus, as for the CHAR type, it has a maximum length of 32K.

While the length of a MIXED data item is declared in terms of single-byte characters, any string functions with MIXED arguments work on actual characters, whether double- or single-byte.

Since the field size is defined in terms of single-byte characters, the number of double-byte characters that can be stored in such a field is a maximum of half the field's specified size. Note that for Unicode based platforms, such as Java, this limitation on double-byte characters is not present. This can lead to different behaviors for such platforms. To minimize such issues, user input fields can optionally be validated for length as detailed in the DBCS Programming chapter of the *Developing Applications Guide*.

User input data is validated for length to ensure that no truncation of non-space characters occurs when undergoing codepage conversion, such as when marshalled by JNetE. Such length validation must occur in the context of a specified codepage, which is typically the one used by JNetE when marshalling data for transmission to backend systems. This is done in order to handle conversions from Unicode to other codepages, and stateful codepages for DBCS regions where characters or escape sequences are embedded in the data to switch modes, thus expanding the data.

When mapping to or constructing a new field of this type, the source data is truncated if it is too long to fit in the destination field. Any such truncation is based on the actual field lengths and does not take into account any length changes that might occur due to a codepage conversion, such as when making remote rule calls. If such truncation causes the second byte of a double-byte character to be truncated, the entire double-byte character is truncated, and the field is padded as necessary.

A MIXED field is padded to its declared length with single-byte spaces. Trimming MIXED data removes both trailing single-byte or double-byte spaces.

Because of the differences in character representation on different platforms, a varied number of characters can fit into a particular MIXED field. Keep the following in mind when writing multi-platform applications:

- DBCS and MIXED Data Types in Java
- DBCS and MIXED Data Types in COBOL (for ClassicCOBOL and OpenCOBOL)

Data Type Conversions

A data item of a certain data type can be converted to a different data type, either explicitly using a conversion function such as DECIMAL(char) or DBCS(mixed), or implicitly. An implicit conversion is performed automatically when a data item of one type is assigned to a variable of another type and the conversion is possible. When such an assignment is performed, and if there is a possibility of data loss because of the data type conversion, a warning is issued for MAP and SET statements.

A data type conversion is also performed for implicit assignments in which actual parameters are passed to a procedure or method. When an implicit conversion is supplied for actual procedure parameters, the same rule applies as for a MAP or SET statement; however, no warnings are generated for the possibility of data loss caused by the data type conversion. Refer to ObjectSpeak Conversions in Java for possible conversions when calling a method.

The implicit conversions are:

- Identity conversions
- Implicit numeric conversions
- Implicit character conversions

An *identity conversion* transforms an expression of any type to a field to the same type.

The *implicit numeric conversions* are:

- Conversion of any numeric expression value to any numeric type.
- Conversion of any numeric expression value to unsigned picture. This case causes warning generation because value of expression can be negative.

The *implicit character conversions* are:

- Conversion of character expression value to character or varchar.
- Conversion of text or image field to character or varchar.
- Conversion of any character expression value to text or image.
- Conversion of unsigned picture to any character type. Such conversion is legal if and only if the unsigned picture is defined as PIC '9...9'.
- Conversion DBCS expression to MIXED.
- Conversion DBCS or MIXED literal to character or varchar.

Platform Specific Consideration

For platform specific considerations, see Implicit Numeric Conversions in Java and Implicit Numeric Conversions in COBOL.

Data Items

A data item — or data element — is an individual unit of data that is processed by a rule. A data item, such as a field, is defined for processing purposes and might have a specific size, type, and range. Views and fields are data items that are named storage locations capable of containing data that can be modified by rules during the program execution. Literals and symbols, on the other hand, are data items that have a constant value throughout the program execution. They can be altered only by manually changing their values within the coding of a rule.

If a field is modified by a component outside of AppBuilder, the component must ensure that the field complies with the AppBuilder definition. If a user component initializes or pads a field, it must perform as if within AppBuilder. Even if the field type has different characteristics in the language the user component is written in, the field must conform to what AppBuilder uses. Refer to Data Types for definitions of data types supported in AppBuilder.

The following data items are described in this section:

- Variable Data Item
- View
- Character Value
- Numeric Value
- Symbol
- Alias

The following table shows how the various data items are classified.

Classification of Data Items

Data Item Name	Variable Data Item	Constant Data Item
View	X	
Field	X	
Symbol		X
Literal		X
Array	X	
Default Object		X
Alias		X

Variable Data Item

A rule can use any view or field in its data universe as a variable data item. You can either define a variable in your repository to be used globally or declare it within a rule to be used locally within that rule. See Local Variable Declaration for information about how to declare a variable locally.

The variables in an application are initialized according to their data types prior to program execution to prevent them from processing unpredictable values. See Initializing Variables for more information.

AppBuilder does not validate non-initialized variables before they are used.

Generally, a view can be a variable data item almost everywhere a field can be; exceptions are noted where they apply. You can transfer data from all the fields of one view directly to all the fields of another view by mapping the first view to the second.

Variable Data

variable_data_item:

field_name (OF *view*)* [‘(‘ index_list ‘)’]

view [‘(‘ index ‘)’] . (*view* [‘(‘ index ‘)’] .)* *field_name*

object_name (. object_speak_reference)*

index_list:

numeric_expression (, numeric_expression)*

index:

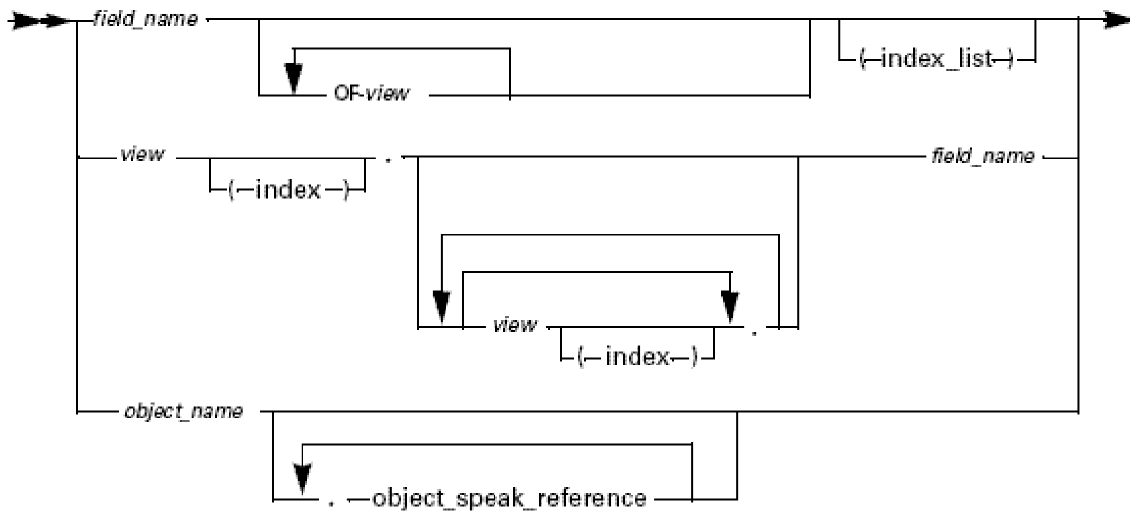
numeric_expression

object_speak_reference:

. property_name

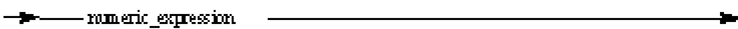
. '(' expression (, expression)* ')'

Variable Data Syntax

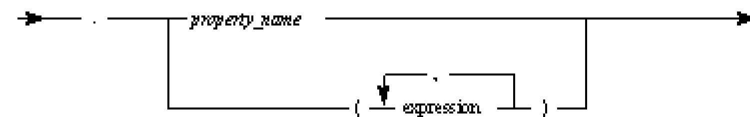


where index_list is:

where index is:



where object_speak_reference is:



where *object_name* can be one of the following:

- The system identifier (HPSID) of the object.
- The alias of the object — see Alias.
- An object — see Object Data Types.
- An array — see Array Object.

where:

- expression — see Expression Syntax.
- numeric_expression — see Numeric Expressions.
- *view* — see View.

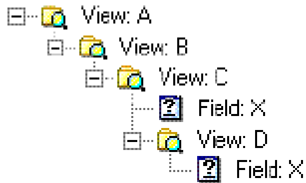
Qualifying Fields

When you reuse an entity defined in the repository, a given field or view can appear more than once in the rule's data universe. In such a case,

referring to just the name of a field or view could lead to an identification conflict. To avoid this, qualify potentially ambiguous references with some or all of the names of the ancestor views of the variable.

Ambiguity in the data hierarchy of a rule is not checked until that rule is prepared. Thus, ambiguity errors will only be issued during the rule preparation. Such an error is usually reported with a BINDFILE prefix in the error message and is issued when a view is used as a top-level view and as a child of another view.

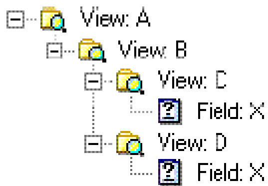
For example, in the following view hierarchy, the View D is a top-level view and also a child of View C. A reference to the variable X becomes ambiguous because there are two occurrences of variable X, and the full path to one of the occurrences is exactly the same as the beginning of the path to the second occurrence.



The following qualifications are ambiguous, because *A.B.C* is exactly and fully included in the path *A.B.C.D.X*:

A.B.C.X
A.B.C.D.X

Consider a different hierarchy as shown below:



The following qualifications are not ambiguous:

A.B.C.X
A.B.D.X

However, the following qualification uses only partial qualification. It is an ambiguous reference:

A.B.X

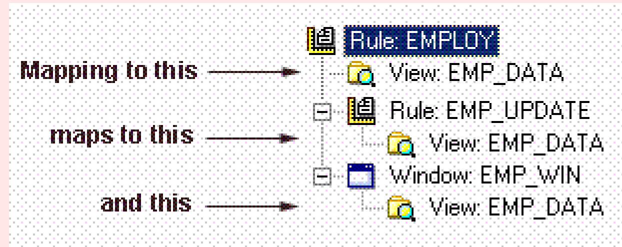
There are two ways to qualify fields: using either "." (dot) notation or the *OF* clause.

To uniquely identify different instances of a field or view, place the name of a containing view before "." and the name of a field or view after it.

Using the OF clause produces the same result. Add the name of an ancestral view in an OF clause following the name of the variable. This type of fields and views qualification differs in order, or direction: when using the OF clause, the sequence begins from the innermost item — a field or view; with the "." (dot) notation, the sequence begins from the outermost item.

You do not have to give the entire list of ancestor or successor views if a partial list uniquely identifies the intended view or field. To be sufficient, the ancestor list must contain the name of at least one view that uniquely identifies the intended view or field.

Be careful when reusing views. A root view is a view whose parent is not another view. The AppBuilder environment considers all root views with the same name in the data universe of a rule to be the same view. (See "Data Universe" topic in the *Developing Applications Guide*.)



Mapping information to one such view maps the information to all root views with the same name in the data universe of a rule, *even if the names are fully qualified*. For example, in the hierarchy shown above, if you map information to EMP_DATA of the rule EMPLOY, that information also appears in two other instances of EMP_DATA, assuming that EMP_DATA under EMP_UPDATE is an input/output view, and not a work view.

Both of the following examples can be used to qualify fields. They each refer to the same fields and are completely equal in rights.

Examples: Field Qualifications and Using Subscripts

The following is an example of field qualifications:

```
*> Using OF clause <*
LAST_NAME OF CUSTOMER OF ALL_CUSTOMERS
LAST_NAME OF ALL_CUSTOMERS

*> Using dot notation <*
ALL_CUSTOMERS.CUSTOMER.LAST_NAME
ALL_CUSTOMERS.LAST_NAME
```

The following examples illustrate the use of subscripts (indexes):

```
*> Using OF clause <*
LAST_NAME OF CUSTOMER OF ALL_CUSTOMERS OF DEPARTMENT(5, 10)

*> Using dot notation <*
DEPARTMENT.ALL_CUSTOMERS(5).CUSTOMER(10).LAST_NAME
```

You can also write:

```
LAST_NAME OF ALL_CUSTOMERS(5, 10)
LAST_NAME(5, 10)
```

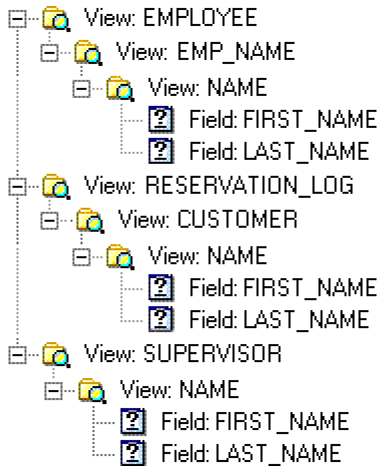
You can omit intermediate views that are not ambiguous. You *cannot* omit views that require indexes when using the dot notation to qualify fields. For example, the following is not correct:

```
DEPARTMENT. (5). (10).LAST_NAME
```

Using Partial Qualification

In the following sample view, the subview NAME is used in three places to hold three categories of data: the employee's name, the employee's customer's name, and the employee's supervisor's name.

Sample View Hierarchy



The following example shows a method of specifying the unique data contained in each field by qualifying a part of an argument based on a unique identifier that precedes the field.

For a fully-qualified statement, the following MAP statement defines the unique identity of:

the employee's LAST_NAME field:

```
MAP 'Attonasio' TO LAST_NAME OF NAME OF EMP_NAME OF EMPLOYEE
```

or the customer's:

```
MAP 'Borges' TO LAST_NAME OF NAME OF CUSTOMER OF
RESERVATION_LOG OF EMPLOYEE
```

or the supervisor's:

```
MAP 'Calvino' TO LAST_NAME OF NAME OF SUPERVISOR OF EMPLOYEE
```

These qualifications, however, can be shortened, since each field has at least one unique ancestor. Thus, the three last-name fields could be identified as:

```
LAST_NAME OF EMP_NAME
LAST_NAME OF CUSTOMER (or LAST_NAME OF RESERVATION_LOG)
LAST_NAME OF SUPERVISOR
```

Since only one instance of LAST_NAME exists beneath each of the EMP_NAME, CUSTOMER (or RESERVATION_LOG) and SUPERVISOR views, specifying any of these view names is sufficient to distinguish between any instance of the LAST_NAME field within the EMPLOYEE view.

Initializing Variables

The variables in an application are initialized according to their data types prior to program execution to prevent them from processing unpredictable values. Initialization of a view causes recursive initialization of every field of that view. Different data types are initialized in different ways and variables of different scope are initialized as required during program execution. When variables have been correctly coded, the system automatically initializes them.

Variables are initialized in the following situations:

- The rule and procedure local variables and rule output views are initialized every time a rule is called and before the rule code is

executed.
- The input view of a rule is initialized in the parent rule.
- Global and all other views are initialized only one time upon application start (main rule start).

The following table shows the data types and the initial values when they are initialized:

Data types and initial values

Data Type	Initialized with...
BOOLEAN	FALSE
CHAR	single-byte spaces
VARCHAR	In C and Java: zero length string In ClassicCOBOL and OpenCOBOL: The character portion is initialized to single-byte spaces and the numeric portion is initialized to zeros.
DBCS	ideographic blanks (DBCS spaces)
MIXED	single-byte spaces
Variables of numeric data types	zero
Large object data types	zero length string
Object references	null reference
DATE variables	January, 1st, 1 AD
TIME variables	00.00.00.000
TIMESTAMP variables	0000-00-00-00.00.00.000000

For additional information, see Initialization in Java and NULL in Java.

View

A View is an object in the Information Model that defines a data structure you use within your rules. Essentially, it is a "container" for other views and fields.

For detailed information about the Information Model, refer to the *Information Model Reference Guide*.

View Name Syntax

view_data_item:

view_name (OF *view_name*)* ['(' index_list ')']

view_name ['(' index ')'] . (*view_name* ['(' index ')'] .)* *view_name* '(' index ')'

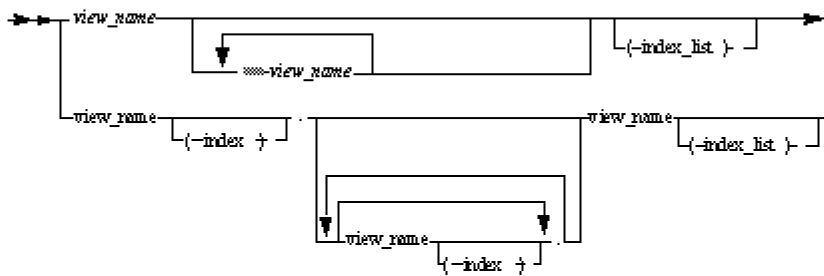
character_value:

symbol

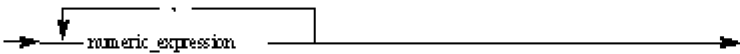
' *string_literal* '

" *string_literal* "

character_field



where index_list is:



where index is:

Usage

When using a view in a Rules Language statement, only the name of the view as it is defined in the repository is required.

You can create what other programming languages call an array (a multiple-accurring vilew) by setting the Occurs property. For more information, see Multiple-Occurring Subview.

By using a view to redefine another view, you can assign data and structure to the redefined view. These two views share the same data in memory without creating two copies of the same data. Refer to Redefining Views for more information.

Platform Specific Consideration

For platform specific considerations, see Views in OpenCOBOL.

View Size Limitations

Generally, AppBuilder has no limitations on the view size; however, some platforms or compilers might have such limitations. Additionally, multiple-occurring views in input/output views for remote rules are limited to 32K occurrences.

For Specific Considerations for ClassicCOBOL and OpenCOBOL, see Size Limitations in ClassicCOBOL and OpenCOBOL.

Multiple-Occurring Subview

Define a *multiple-occurring subview* by changing the Occurs times property in the "View *includes* View" relationship that connects the child view to its parent. This creates what other programming languages call an array. Each "row" is simply an indexed instance of the child view, with the same field data types. The Occurs property works with the "View *includes* View" relationship only. This means that neither fields nor top-level views can be subscripted.

You can refer by number to a specific instance of a multiple-occurring subview within the rule that owns the including view. To reference an individual occurrence of an item in a multiple-occurring view, place the occurrence number of the multiple-occurring view in parentheses after the last qualifier. This index can be any expression that resolves to a number. Because occurrence numbers can only be integers, any fractional part to an index value will be truncated.

As discussed in Variable Data Item, you can omit the names of some of a field's ancestral views in a statement. However, you must always include the occurrence number of a multiple-occurring subview in a statement, even if the view's name does not appear.

Arrays can be nested to a maximum of three levels. To refer to a particular element, list the indices of its parent views in parentheses after the last qualifier.

When using the OF clause, while the view names ascend in the hierarchy when reading from left to right, you must place the subscripts in reverse order so that they descend in the hierarchy. When using the dot notation, the subscripts are in the same order as the view names are listed in the statement.

For OpenCOBOL, you can prevent an occurring view from being initialized. For more information, see Initialization of Occurring Views in OpenCOBOL.

For specific considerations for Java, see Dynamically-Set View Functions in Java.

Example: Using the Occurs Property to Set the View

Assume the relationship between a view named DEPARTMENT and its subview named EMPLOYEE has an Occurs property of 20, and EMPLOYEE view contains a field LAST_NAME. Thus, each DEPARTMENT view has 20 EMPLOYEE views, each of which can be accessed separately by any rule that has the DEPARTMENT view in its data universe. The following statements reference the twelfth occurrence of the employee view:

```
MAP 'Jones' TO LAST_NAME OF EMPLOYEE OF DEPARTMENT(12)
```

or

```
MAP 'Jones' TO DEPARTMENT.EMPLOYEE(12).LAST_NAME
```

Assume a SITE view includes the DEPARTMENT view, and the DEPARTMENT view is itself occurring. The following statements reference the twelfth employee's last name in Department 4:

```
MAP 'Jones' TO LAST_NAME OF DEPARTMENT OF SITE (4,12)
```

or

```
MAP 'Jones' TO SITE.DEPARTMENT(4).EMPLOYEE(12).LAST_NAME
```

Even though the name of the EMPLOYEE view does not appear in the statement, you must include the subscript for this view to avoid ambiguity.

Use caution with regard to the fields because they can be unqualified in the rule code. Statements like the following can be confused for subscripted fields:

```
MAP 'Jones' TO LAST_NAME(12)
```

The subscript here does not refer to the LAST_NAME field, but to the omitted EMPLOYEE view.

Character Value

A character value can be a symbol associated with a character value, a field of a character data type, or a character literal. A character literal is a string of up to 50 characters enclosed in single or double quotation marks.

Character Value Syntax

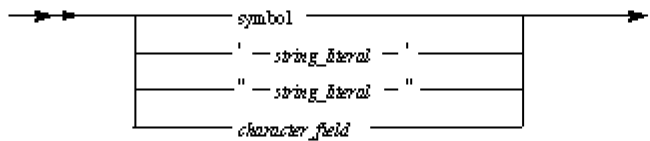
character_value:

symbol

‘ *string_literal* ’

“ *string_literal* “

character_field



where:

- *character_field* is a variable data item of any character type.
- symbol — see Symbol.

Using Character Literals

Character literals can be enclosed in double or single quotation marks. If a literal begins with a single quotation mark, it must end with a single quotation mark. If a literal begins with a double quotation mark, it must also end with a double quotation mark. A string literal is a constant.

You can include a single-quotation mark (') as part of a character string by putting two single quotation marks (not a double quotation mark) in its place in the string. For example,

```
MAP 'Enter the item''s price' TO message
```

Two consecutive single quotation marks (when not being used as part of a character string) represent a null string, which is the value of an empty character field.

Character literals can be placed on one or more lines of Rules Language code. To place the literal on several source code lines, put quotation marks at the end of the first line and begin the second line with the same quotation mark. For example:

```
MAP 'Orson Welles,'
    'Frank Capra,'
    'Preston Sturges.' TO director_list
```

or

```
MAP "Orson Welles,"
    "Frank Capra,"
    "Preston Sturges." TO director_list
```

Example: String Literal

The following is an example of a string literal in a MAP statement:

```
MAP 'This is a character literal' TO MESSAGE
```

The MAP statement

```
MAP 'Enter the items''s price' TO MESSAGE
```

puts the string "Enter the item's price" in the MESSAGE field.

The following statement tests for an empty value in a character field:

```
IF NAME = ''
```

Using Escape Sequences within Literals

The backslash (\) starts an escape sequence and is valid only in literals within double quotation marks. These escape sequences allow you to use a sequence of characters to represent special characters, including non-printing characters. Escape sequences also allow you to specify characters with hexadecimal or octal notation. The supported escape sequences are listed in the following table.

Supported Escape Sequences

Escape Sequence	Character Name
\a	Alert (Bell)
\b	Backspace
\f	Form feed
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\?	Question mark
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\o...o	octal number
\xh...h	hexadecimal number

DBCS and MIXED Literals

Both single-quotation mark and double-quotation mark literals can contain Double Byte Character Set (DBCS) characters. If such a literal contains only DBCS characters, it is considered a DBCS literal. If a literal contains both DBCS and Single Byte Character Set (SBCS) characters, it is considered a MIXED literal.

Using Hexadecimal and Octal Notation

Each hexadecimal or octal number represents only one character. Therefore, if an octal or hexadecimal number is greater than 255, the character value is equal to the remainder of integer division of this number by 256.

For characters notated in hexadecimal, leading zeros are ignored. During preparation, the code generation process establishes the end of the hexadecimal or octal number when it encounters the first non-hexadecimal or non-octal character correspondingly. For example, *\x0cDebug* will have only three characters: *\xEB*, *u*, and *g*.

To define two bytes, use the string literal *\x11\x10*. This is converted to two bytes, the first with a decimal value of 17 and the second with a decimal value of 16.

Example: Hexadecimal Notation

- */x10* - This is not a hexadecimal value because it begins with a forward slash, instead of a backslash.
- *\x10* - This is converted to one byte, and equates to a hexadecimal value of 10, a decimal value of 16, and a binary value of 00010000.
- *\x1110* - This is converted to one byte, and equates to a hexadecimal value 10, a decimal value of 16 (the remainder of integer division 4368 by 256), and a binary value of 00010000.

Numeric Value

A numeric value can be a symbol associated with a numeric value, a field of a numeric data type, or a numeric literal. A numeric literal is either an

integer or a decimal number.

Numeric Value Syntax

numeric_value:

symbol

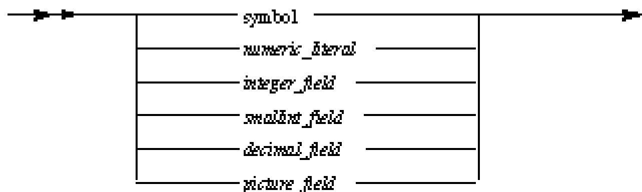
numeric_literal

integer_field

smallint_field

decimal_field

picture_field



where:

- symbol — see [Symbol].
- *integer_field* is a variable data item of INTEGER data type.
- *smallint_field* is a variable data item of SMALLINT data type.
- *decimal_field* is a variable data item of DEC data type.
- *picture_field* is a variable data item of PIC data type.

Numeric Literals

An integer literal must be within the range specified for the data type. See SMALLINT and INTEGER for allowed ranges. A decimal literal can be no more than 31 digits long, regardless of the position of the decimal point within the number. To denote a negative number, precede a numeric literal with a minus sign (–). A numeric literal is a constant.

There are two types of supported numeric literals — decimal and hexadecimal. Decimal literals are a sequence of decimal digits. The integer and fraction components of the literal are separated by a period.

Integer and decimal numbers can be represented as hexadecimal literals: *0xFF* represents 255. Hexadecimal literals begin with *0x* or *0X* characters, which are followed by *n* hexadecimal digits (0 < *n* 29). For example, the following literals are all equal:

- 255
- 0xff
- 0Xff
- 0x0ff
- 0XFF
- 0x00FF

See Using Hexadecimal and Octal Notation for additional information about hexadecimal usage.

For an example of associating a symbol with a numeric value, see Example: Symbols in Rules.

Examples: Integer and Decimal literals

An integer numeric literal: 42
A decimal numeric literal: -324.85

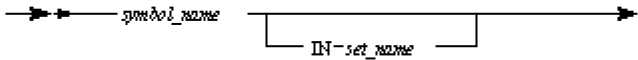
Symbol

Because the value of a symbol does not change during program execution, it is a constant. You can store character and numeric literals in the repository as symbol entities and group them into sets. You can use symbols to specify "define," "encoding," and "display." Rules normally refer to set symbols by the "define."

Symbol Syntax

symbol:

symbol_name [IN *set_name*]



Usage

To use a symbol in a rule, the rule must have a "refers-to" relationship with the set that contains the symbol.

A special case arises when a symbol is used in a CASE statement and the symbol has the same name as a rule. In this case, it is ambiguous whether the CASE statement is referring to the symbol or to the rule. To prevent ambiguity, enclose the symbol name within parentheses:

```
CASE (symbol_name IN set)
```

For additional information regarding the use of Symbol in OpenCOBOL, see Symbols in OpenCOBOL.

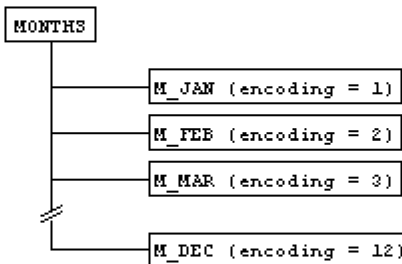
Using IN Clause

If the name of a symbol appears more than once in the data universe of a rule (either in another set or as a field or view name), you must specify the name of the set in an IN clause.

Example: Symbols in Rules

Assume a set MONTHS, as shown in the following figure, has 12 member symbols with the "encoding" (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12) and the "defines" (M_JAN, M_FEB, M_MAR, M_APR, M_MAY, M_JUN, M_JUL, M_AUG, M_SEP, M_OCT, M_NOV and M_DEC).

Set MONTHS



A rule referring to the MONTHS set can use the symbol M_JAN exactly as if it were the number 1, the symbol M_FEB exactly as if it were the number 2, and so on:

```
DO FROM M_JAN TO M_DEC INDEX CURRENT_MONTH
  MAP YEARLY_TOTAL + MONTHLY_TOTALS (CURRENT_MONTH) TO YEARLY_TOTAL
ENDDO
```

Alias

An alias is a name assigned to a data item of OBJECT data type to be used in the Rules Language in place of a system ID to refer to an object. Declare an alias when:

- The system ID is not a valid Rules Language Identifier – a valid Rules ID contains only alphanumeric characters or underscores, with no empty spaces.
- The system ID is the same as an existing field, view, set, symbol, or keyword.

Alias Syntax

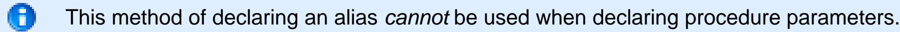
alias:

alias_name OBJECT ' *system_identifier* '

alias OBJECT 'system_identifier'

where

- *alias* is any valid Rules Language identifier.
- *system_identifier* is the system identifier of an object declared in the panel file.

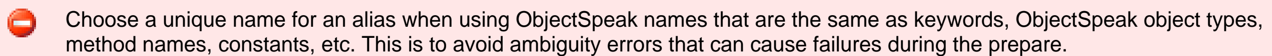


This method of declaring an alias *cannot* be used when declaring procedure parameters.

The system ID in an alias declaration is a character literal, and is case-sensitive. The system ID in the declaration must exactly match the system ID as entered in Window Painter in the property page of a window object.

The object must exist in a window used by the rule in which the object is declared.

Once an alias is declared, the original name is no longer available for use. *Only* the alias can be used to refer to the object.



Choose a unique name for an alias when using ObjectSpeak names that are the same as keywords, ObjectSpeak object types, method names, constants, etc. This is to avoid ambiguity errors that can cause failures during the prepare.

For information about the default object, refer to OBJECT and OBJECT POINTER in Java.

Example: Declaring an Alias

The following example declares "Button1" as the alias for an OBJECT type data item, which has the system ID "Button_1".

```
DCL
   Button1 OBJECT 'Button_1';
ENDDCL
```

Expressions and Conditions

An expression is any Rules Language construct with a character or numeric value. Any field, symbol, literal, or function that evaluates to a specific value is an expression. A view is also considered as an expression. A complex expression is two expressions joined by an operator.

The order of expression and condition evaluation is not guaranteed to be the same as it is written. This is important because the evaluation of some operands of the expression or condition can lead to unexpected results such as the modification of global variables. For example:

```
flag = TRUE OR my_date < my_proc(date)
```

In the above example, the second condition (my_date < my_proc(date)) might be evaluated before the first condition (flag = TRUE). If my_proc changes the value of the flag variable, the order of evaluation might lead to different results. See Example: Order of Expression and Condition Evaluation for an example in which the order of the expression and condition evaluation is different from the order they are written.

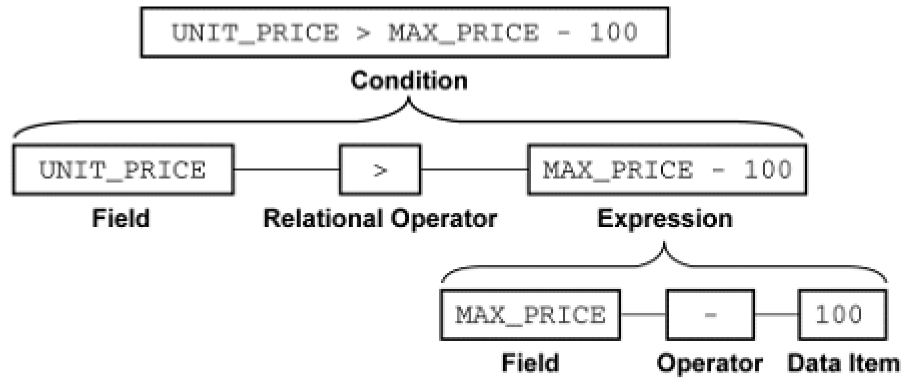
The following topics are discussed in this chapter:

- Character Expressions
- Numeric Expressions
- DATE and TIME Expressions
- Arithmetic Operators
- Condition Operators
- Comparing Fields with Expression

Because methods can return a value, the method invocation can be a constituent of an expression. For information about invoking a method, see ObjectSpeak Statement.

Conditions for Rules Language Statements shows how conditions and expressions can be built from data items.

Conditions for Rules Language Statements



Expression Syntax

expression:

character_expression

numeric_expression

object_expression

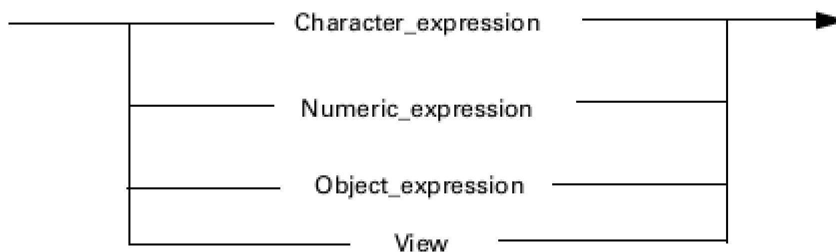
date_expression

time_expression

timestamp_expression

boolean_expression

view_expression



Example: Order of Expression and Condition Evaluation

In the following example, the procedure P is executed before the "I-P" expression, thus the result is different from what you might expect:

```
DCL
   I integer;
ENDDCL

PROC P : INTEGER
   MAP I + 1 TO I
   PROC RETURN (I)
ENDPROC

MAP I - P TO I
TRACE(I) // 0 will be printed, while one may expect -1
```

This is because P is executed before I-P is evaluated.

Character Expressions

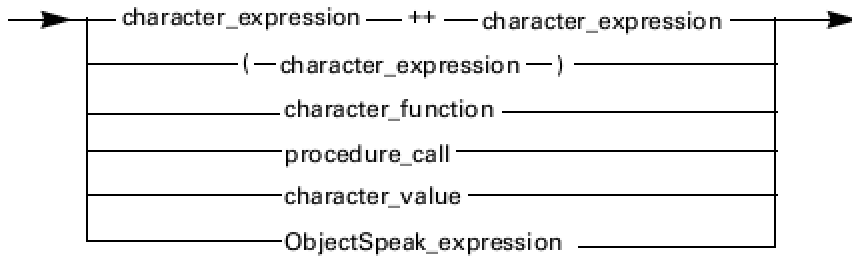
character_expression:

character_value

character_function

procedure_call

objectspeak_expression



where:

- ++ is the concatenation of two expressions — see ++ (Concatenation).
- character_function — see Character String Functions.
- procedure_call — see PERFORM Statement.
- character_value — see Character Value.
- ObjectSpeak_expression — see ObjectSpeak Statement.

Numeric Expressions

numeric_expression:

numeric_value

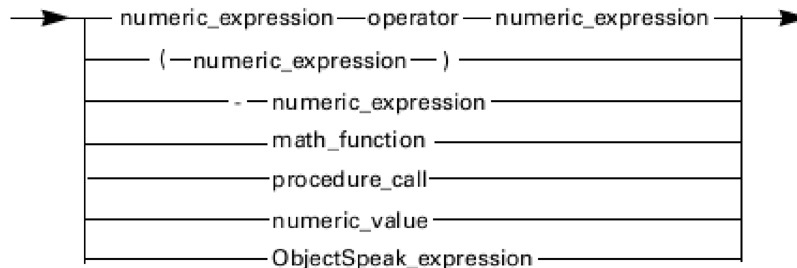
numeric_function

procedure_call

- numeric_expression

numeric_expression arithmetic_operator numeric_expression

ObjectSpeak_expression



where:

- operator — see Arithmetic Operators.
- math_function — see Mathematical Functions.
- procedure_call — see PERFORM Statement.
- numeric_value — see Numeric Value.
- ObjectSpeak_expression — see ObjectSpeak Statement.

Using – Expression (unary minus)

The *unary minus* is the shorthand for

0 - *expression*

as in

```
MAP - (HOUR OF MILITARY_TIME MOD 12) TO A
```

To avoid having two operators in a row, put parentheses around any expression in this format if an operator immediately precedes it:

```
A MOD (- B)
```

instead of

```
A MOD - B
```

Parentheses in a complex expression override the normal order of operations. (See Arithmetic Operators, for that order.) When there is more than one set of parentheses in an expression, left and right parentheses are matched and resolved from the inside out.

Example: Expressions

Both *12* and *7* are numeric literals and hence are expressions; *12 + 7* is a complex expression that combines the two numeric literals with the addition operator (+) to represent the value 19.

Similarly, the following are all expressions:

```
104366564 + 14223412
PRICE OF ITEM_1 * TAX
HOUR OF MILITARY_TIME MOD 12
ROUND(12.47)
```

Because an expression can be used again in the expression definition as a sub-expression, these are also expressions:

```
104 + 23 * 3
(104 + 23) * 3
PRICE OF ITEM_1 + PRICE OF ITEM_1 * TAX
60 * (HOUR OF MILITARY_TIME MOD 12)
ROUND(5.389) * 10 + 2
```

Expressions can be used to generate even more complex expressions:

```
(104 + 23 - 3) DIV 2
```

Note that because of the parentheses, this is equivalent to 124 DIV 2. Similarly, in the statement

```
(10 * (5-2))
```

the inside set of parentheses is first resolved to 3, and then the outside set is resolved to 30.

DATE and TIME Expressions

Use data items of DATE and TIME data types in numeric expressions with certain limitations:

- "-" (unary minus) is not allowed.
- Do not use mixed combinations of the DATE and TIME operands. If one of the operands is DATE (TIME), then the other must be either numeric or of the same data type - DATE (TIME).

No arithmetic operations with data items of TIMESTAMP data type are supported.

The following table represents valid combinations of operands and the type of result for each combination.

Valid Operands

Left operand type	Operator	Right operand type	Result type
DATE (TIME)	+, -, *, /, **, DIV, MOD	DATE (TIME)	Numeric
DATE (TIME)	+, -	Numeric	DATE (TIME)
DATE (TIME)	*, /, **, DIV, MOD	Numeric	Numeric
Numeric	+	DATE (TIME)	DATE (TIME)
Numeric	-, *, /, **, DIV, MOD	DATE (TIME)	Numeric

When using DATE or TIME in arithmetic expression, its *value* is used, i.e. number of days past the date of origin for DATE and the number of milliseconds past midnight for TIME. Use the INT function (see INT) to obtain the *value* of data items of DATE or TIME data type.

Precision of the expression with DATE or TIME is calculated with the assumption that the DATE and TIME value has type INTEGER.

Example: Using DATE and TIME Expressions

The following are examples of expressions using DATE and TIME data types

```
DCL
    DT DATE;
    TM TIME;

    I INTEGER;
    SM SMALLINT;
ENDDCL

MAP 1 TO SM
MAP 1000 TO I

MAP DATE ('05/03/99', '%0m/%0d/%0y')     TO DT // 05/03/1999
MAP TIME ('1:22:03 PM', '%h:%0m:%0s %x') TO TM // 13:22:03:000

MAP DT + 1     TO DT // 05/04/1999 - next day
MAP SM + DT    TO DT // 05/05/1999

MAP TM + 1000 TO TM // 13:22:04:000 - next second
MAP I + TM     TO TM // 13:22:05:000

MAP DT - 1     TO DT // 05/04/1999 - previous day
MAP TM - 1000 TO TM // 13:22:04:000 - previous second

MAP INT(DT)    TO I // 730244
MAP DT + DT    TO I // 1460488 (730244 + 730244)
MAP DT / 2     TO I // 365122 (730244 / 2)

RETURN
```

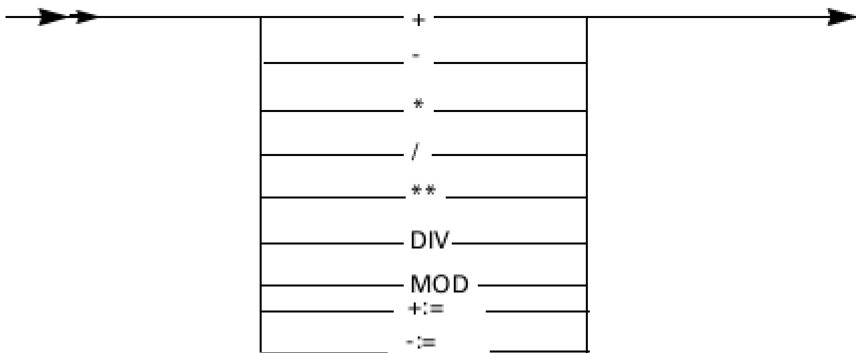
Arithmetic Operators

The Rules Language supports the following basic arithmetic operations. Write and call your own user components if you need to perform more complex calculations. (For more information about user components, refer to the *Developing Applications Guide*.)

Arithmetic Operator Syntax

arithmetic_operator:

one of + - * / ** DIV MOD +:= -:=



Generally, the precision of an arithmetic operation is controlled by the data item in which the result is stored. For instance, the statement

```
MAP 2001 / 100 TO X
```

produces a value of 20 if X is declared as SMALLINT or INTEGER, a value of 20.0 if X is declared as DEC (3, 1), and a value of 20.01 if X is declared as DEC (4, 2) or larger.

When you perform an addition, subtraction, or multiplication operation on two SMALLINT data items, the result is also a SMALLINT data item, regardless of the actual type of the target variable. That is, the result is calculated before the map operation takes place. This could cause an overflow.

For example, *MAP SMALLINT_VAR * SMALLINT_VAR TO INTEGER_VAR* overflows if the product of the two SMALLINT data items is greater than 32,767.
If you expect the result to be outside the range of SMALLINT data type, use INTEGER or DEC data type.

Operator Precedence

As in most programming languages, the *, /, MOD, and DIV operators take precedence over the + and - operators. Also, unary minus and the exponential operator take precedence over *, /, MOD, and DIV. You can use parentheses in the expression syntax to override the order of operations. All the arithmetic operators take precedence over the relational and Boolean operators.

The following table summarizes the order of precedence of the arithmetic operators. (Unary minus is discussed under Using – Expression (unary minus).)

Arithmetic operator precedence

Operator	Precedence
unary minus, **	highest
*, /, MOD, DIV	↓
+, -	↓
+:=, -:=	lowest

See the following for more information about each operator:

- ++ (Addition)+
- - (Subtraction)
- * (Multiplication)
- / (Division)
- ** (Exponentiation)
- DIV (Integer division)
- MOD (Modulus)
- +:= (Increment)
- -:= (Decrement)

+ (Addition)

The arithmetic operator + adds two expressions together. The statement

```
MAP 10 + 4 TO X
```

gives X a value of 14.

- (Subtraction)

The arithmetic operator – subtracts its second expression from its first. The statement

```
MAP 10 - 4 TO X
```

gives X a value of 6.

* (Multiplication)

The arithmetic operator * multiplies two expressions. The statement

```
MAP 10 * 4 TO X
```

gives X a value of 40.

/ (Division)

The arithmetic operator / divides its first expression by its second. The result might be an integer or a decimal number depending on how the variable is declared, which holds the result. The statement

```
MAP 10 / 2 TO X
```

gives X a value of 5.

The statement

```
MAP 10 / 3 TO X
```

gives X a value of 3.33333333333333, assuming X was declared as DEC(15, 14).

See also DIV (Integer division), which always returns an integer.

** (Exponentiation)

The arithmetic operator ** raises its first expression to the power of its second. The first expression can be any numeric type, but the second

expression must be a SMALLINT or INTEGER on the mainframe and any numeric type on the workstation. The statement

```
MAP 10 ** 4 TO X
```

gives X a value of 10,000. The statement

```
MAP 3.14 ** 2 TO X
```

gives X a value of 9.8596.

Note that the result of the exponentiation operation might be too large to represent in the largest data type, DEC (31). If the first expression of exponentiation operation is negative and the second expression is a fractional number then invalid exponentiation error message is generated.

DIV (Integer division)

The arithmetic operator DIV returns the number of times the second operand can fit into the first. Both operands can be decimal numbers, but the result is always an integer. The statement

```
MAP 11 DIV 2 TO X
```

gives X a value of 5, since 2 fits into 11 five times (with a remainder of 1).

The following are examples of the DIV operator:

`MAP 11 DIV 0.2 TO X`	gives X a value of 55
`MAP 1.1 DIV 0.2 TO X`	gives X a value of 5
`MAP 0.11 DIV 0.2 TO X`	gives X a value of 0

See also / (Division), which can return either an integer or a decimal number.

MOD (Modulus)

The arithmetic operator MOD provides the remainder from an integer division operation. The statement

```
MAP 11 MOD 2 TO X
```

gives X a value of 1, since 2 fits into 11 five times, and the remainder (modulus) is 1.

If MOD is declared as a decimal, it returns a decimal remainder when the remainder is not a whole number. If MOD is declared as an INTEGER or SMALLINT, it will not return a decimal remainder.

The following are examples of the MOD operator

`MAP 11 MOD 0.2 TO X`	gives X a value of 0

`MAP 1.1 MOD 0.2 TO X`	gives X a value of 0.1
`MAP 0.11 MOD 0.2 TO X`	gives X a value of 0.11

+:= (Increment)

The arithmetic operator +:= adds the right operand to the variable, which is its left operand. The result is the value of this variable. The left operand of this operator must be a numeric variable. The right operand must be a numeric data item. For example:

```
MAP 1 TO I
MAP I +:= 1 TO J    *> sets I and J to 2 <*
MAP I +:= 1-1 TO J  *> I and J are left unchanged because the Increment
                       operator has the lowest precedence and this
                       statement is equal to MAP I +:= 0 TO J <*
```

The increment operator modifies its left operand. Because the order of calculation varies on different platforms, the same expression can give different results on different platforms. Use this operator with caution.

-:= (Decrement)

The arithmetic operator -:= subtracts its right operand from the variable, which is its left operand. The result is the value of this variable. The left operand of this operator must be a numeric variable. The right operand must be a numeric data item. For example:

```
MAP 1 TO I
MAP I -:= 1 TO J    *>sets I and J to 0<*
MAP I -:= 1-1 TO J  *> I and J are left unchanged because Decrement
                       operator has the lowest precedence and this
                       statement is equal to MAP I +:= 0 TO J <*
```

The decrement operator modifies its left operand. Because the order of calculation varies on different platforms, the same expression can give different results on different platforms. Use this operator with caution.

Condition Operators

A condition is an expression that evaluates to either true or false. A condition can be one of two types:

- Relational Condition
- Boolean Condition

Condition Operators Syntax

condition:

relational_condition

boolean_condition

relational_condition:

expression relational_condition_operator expression

expression INSET *set_name*

ISCLEAR *view_name*

ISCLEAR *field_name*

relational_condition_operator:

one of = <> < <= > >=

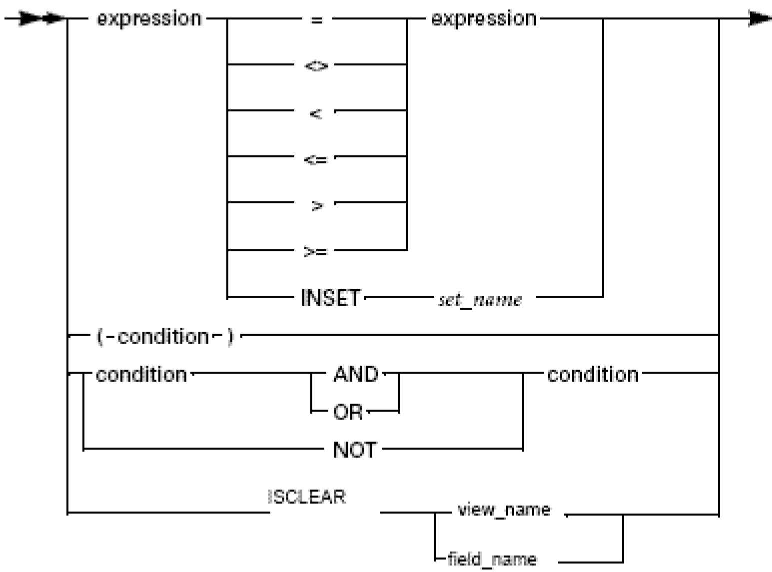
boolean_condition:

condition boolean_condition_operator condition

NOT condition

boolean_condition_operator:

one of AND OR



where:

- expression — see Expression Syntax.

Order of Operations

The order of operations for the relational and Boolean operators, in decreasing precedence, is shown in the following table. All the arithmetic operators take precedence over the relational and Boolean operators. As with expressions, parentheses override the usual order of operations.

Relational and Boolean Operator Precedence

Operator	Precedence
INSET, ISCLEAR	highest
=, <>, <, <=, >, >=	
NOT	
AND	
OR	lowest

Relational Condition

A relational condition compares one expression to another. It consists of an expression followed by a relational operator, then followed by another expression, except for ISCLEAR Operator or INSET Operator. The two expressions must resolve to compatible values. If either expression is a variable data item, the condition's truth or falsity depends on the value of the data item.

The following table lists the relational operators, the comparisons, and the data type variables to which each applies.

Relational Operators

Relational Operator	Comparison	Applicable to Data Types
=	Is equal to	Numeric, Character, Date and Time, large object, Boolean, Object, View
<>	Is not equal to	Numeric, Character, Date and Time, large object, Boolean, Object, View
<	Is less than	Numeric, Character, Date and Time, large object, View
<=	Is less than or equal to	Numeric, Character, Date and Time, large object, View
>	Is greater than	Numeric, Character, Date and Time, large object, View
>=	Is greater than or equal to	Numeric, Character, Date and Time, large object, View
INSET	Is included in the set	Set
ISCLEAR	Is operand value equal to its initial value	Any Variable or View

In C, you cannot compare for equality or inequality aliases and variables of type OBJECT ARRAY.
In Java, you can compare any objects for equality or inequality.

A relational condition can be the argument to a flow-of-control statement to allow it to choose among different actions depending on the condition's accuracy. In the code sample below, the IF...ELSE...ENDIF statement checks whether the condition RETURN_CODE OF UPDATE_CUSTOMER_ DETAIL_O = 'FAILURE' is true. If it is, it displays an error message window. If it is not true, it sets the rule's return code to UPDATE.

Example: Relational Condition Code

```
MAP CUSTOMER_DETAIL TO UPDATE_CUSTOMER_DETAIL_I
USE RULE UPDATE_CUSTOMER_DETAIL
IF RET_CODE OF UPDATE_CUSTOMER_DETAIL_O = 'FAILURE'
   MAP 'CUSTOMER_DETAIL_FILE_MESSAGES' TO MESSAGE_SET_NAME OF
   SET_WINDOW_MESSAGE_I
   MAP UPDATE_FAILED IN CUSTOMER_DETAIL_FILE_MESSAGES TO
   TEXT_CODE OF SET_WINDOW_MESSAGE_I
   MAP 'CUSTOMER_DETAIL' TO WINDOW_LONG_NAME OF
   SET_WINDOW_MESSAGE_I
   USE COMPONENT SET_WINDOW_MESSAGE
ELSE
   MAP 'UPDATE' TO RET_CODE OF
   DISPLAY_CUSTOMER_DETAIL_O
ENDIF
```

INSET Operator

Unlike other relational operators, the name of a Set must follow an INSET operator. A condition with an INSET clause is true if the expression on the left evaluates to a value that is equal to a value entity related to the Set on the right, or encoding if the type of the Set is LOOKUP. The data type of the expression must be compatible with that of the value entity, either both numeric or both character.

Assume there is a Set MONTHS in the data universe of a rule that contains the symbols JAN, FEB, MAR, APR, ... , DEC, representing the values 1, 2, 3, 4, ... , 12. In that rule's code, the condition

```
3 INSET MONTHS
```

is true, because the set MONTHS does contain a value entity whose value property is 3: the value with the symbol MAR. The condition

```
26 INSET MONTHS
```

is false, because there is no such member value in MONTHS. The condition

```
'DECEMBER' INSET MONTHS
```

is illegal, because the MONTHS set is numeric and the literal data item DECEMBER is character.

If the Set type is DBCS, the left operand can be either DBCS or MIXED. If the Set type is MIXED, the left operand can be either MIXED, CHAR, or DBCS. If the Set type is CHAR, the left operand can be either MIXED or CHAR. In all other cases, the explicit conversion function is required. The Set types and the left operand types are summarized in the following table.

Allowed left operand types for INSET operator with characters sets

		Set Type		
		CHAR	MIXED	DBCS
Left Operand	CHAR	O	O	requires conversion function
	MIXED	O	O	O
	DBCS	requires conversion function	O	O

Boolean Condition with INSET

Since the subconditions can also be Boolean, a condition can be built up like an expression:

```
IF (A > B AND NOT (C <= D OR E = F)) OR G INSET SET_H...
```

Order of Operations with INSET

The condition

```
NOT A = B AND C INSET D OR E = F
```

is equivalent to

```
((NOT (A = B)) AND (C INSET D)) OR (E = F)
```

You can change this order using parentheses to something else, such as:

```
NOT (A = B) AND (C INSET D OR E = F)
```

ISCLEAR Operator

The ISCLEAR condition returns TRUE if the variables (or all fields of a view) are set to their initial value, and it returns FALSE if a variable (or at least one field in a view) differs from its initial value. A field is considered to be set to its initialized value if it has never been modified by a user or if it has been reset programmatically: for example, by using the CLEAR statement or by moving a zero to a numeric field.

MIXED variables are considered clear if they contain blanks only. It does not matter if the blanks are double-byte or single-byte. See the Initializing Variables for more information.

To test a variable of any object type in Java for null, or a variable of any type except OBJECT ARRAY and aliases in C (see Object Data Types for an explanation of the OBJECT ARRAY data type) for null, use ISCLEAR instead of ISNULL; it returns TRUE if the reference actually refers to nothing (it contains a null value), ant it returns FALSE otherwise.

Before the ISCLEAR was supported, view comparison was sometimes used to check whether or not a view was modified. Because comparison of views has undefined consequences, and changes in the code generation might result in changes in the behavior, do not use view comparison. Use ISCLEAR instead.

Order of Operations with ISCLEAR

The condition

```
NOT ISCLEAR C AND C INSET D OR E=F
```

is equivalent to

```
((NOT (ISCLEAR C)AND(C INSET D))OR(E=F)
```

You can change this order by changing position of the parentheses.

Example: Using ISCLEAR

The following example tests whether or not all fields of *VIEW1* is set to its initial value.

```
IF ISCLEAR(VIEW1) USE RULE RULE1 ENDIF
```

Boolean Condition

A Boolean condition is one of the following:

- two conditions joined by a Boolean operator AND or OR
- a Boolean operator NOT followed by a condition

The conditions in a Boolean condition can be either relational or Boolean.

The true or false value of a Boolean condition is determined by the values of its two conditions. The following are rules of Boolean algebra:

- *condition* AND *condition* is true only if both conditions are true
- *condition* OR *condition* is true only if one or both conditions are true
- NOT *condition* is true only if the condition is false

The result of relational and Boolean conditions has a BOOLEAN type (See BOOLEAN Data Type). This allows the use of conditions in statements not limited to condition statements.

For example, you can store the results of a comparison for later use with the statement:

```
MAP CUSTOMER_NAME OF UPDATE_DETAILS_WND_IO = CUSTOMER_NAME OF CURRENT_CUSTOMER_V TO IsSameCustomer
```

and use the variable *IsSameCustomer* (of type BOOLEAN) later. Storing results is useful if you want to use the same condition several times; however, using a variable to hold the result of a comparison is not the same as the comparison function itself. In the case of variables holding the result of a comparison, the variables are compared only once; if the values change later, the result is not updated. For example, in the case above, *IsSameCustomer* is not updated.

Comparing Fields with Expression

A PIC field can be compared to either a numeric or a character expression. See PIC for more information. Variables of a large object data type (IMAGE or TEXT) can be compared with each other and with character data items (CHAR or VARCHAR) in any combination.

These topics describe restrictions and special considerations when comparing certain data items:

- Comparing Character Values
- Comparing Views
- Identifying Illegal Comparisons

Comparing Character Values

You can compare character values using all the relational operators except ISCLEAR and INSET. Any such comparison is a straight binary comparison, using the collating sequence of the hardware execution platform, either ASCII (on the workstation in C), Unicode (in Java) or EBCDIC (on the host). Thus, because the numeric values of ASCII and EBCDIC alphabetic characters differ, "abc" is less than "ABC" on the host, but greater than "ABC" on the workstation, both in C and Java.

Since 7-bit ASCII is included into Unicode, which is used in Java, you can assume that collation of all strings containing only 7-bit ASCII symbols remains the same.

DBCS (double-byte character set) strings are compared according to system locale settings. In Java, DBCS is translated to Unicode by the Java compiler. A DBCS symbol in one code page might have a different code in another, but Unicode supports virtually all code pages; thus, the character order for DBCS string on the PC might differ from the character order of the same string in Unicode. Consider these differences when preparing and using your application in Java.

Blanks at the end of a string are ignored when comparing that string for equality to another character expression.

For example, if A, B, and C are defined as follows:

```
A = 'Hello'
B = 'Hello   '
C = 'Hello      '
```

then the following conditions are all true:

```
A = B
B = C
C = A
```

When comparing strings using relational operators <, >, <= or >=, the shorter string is padded with spaces (double-byte or single-byte) to the length of the longer string. CHAR values are padded with single-byte spaces; DBCS values are padded with double-byte spaces. For MIXED values, the blanks at the end of each string are removed and the smaller item is padded with spaces corresponding to the type of character in that position in the longer string.

In Java, ClassicCOBOL and OpenCOBOL, you can also compare DBCS and CHAR data types with MIXED values. (DBCS and CHAR values are implicitly converted to MIXED using the corresponding conversion function.) The comparison is performed character to character not byte to byte. When a single-byte character is compared to a double-byte character, the double-byte character is considered greater than the single-byte character.

CHAR and DBCS values can also be compared.

Comparing Views

It is not recommended to use view comparisons.

You can compare views using standard relational operators (<, >, <=, >=, =, and <>). When comparing views with different structures, the results of the view comparison might be different depending on the target language because some data types are represented differently for different languages. For example, the representation of DATE and TIME data types are different for ClassicCOBOL and OpenCOBOL generations. In particular, do not use view comparison to check if a view has been modified; instead, use the ISCLEAR function (see ISCLEAR Operator).

For language specific information see the following:

- Comparing Views in C
- Comparing Views in Java
- Comparing Views in ClassicCOBOL and OpenCOBOL

Identifying Illegal Comparisons

There are some restrictions when comparing different types data as described below:

- A condition (even if it evaluates to true or false) is illegal if it compares two literals, or if it cannot be true due to the data types of its expressions. For instance, 3 < 29, while conforming to the syntactic definition of a condition, is an error because it compares two literals. A condition that compares only a literal to a set symbol is permitted, so IF MARY IN NAMESET = 4... is legal.

  However, MIXED or DBCS literals compared to a literal of any other legal data type using <, >, <= or >= is considered a legal comparison and is performed at runtime; however, a comparison of two CHAR literals is performed at compile time. This is because of possible differences between compile-time and runtime codepages.

- You cannot compare a SMALLINT field to an expression with a value greater than 32,767 or less than -32,767. If SALARY were a field of data type SMALLINT, then the condition SALARY > 1,000,000 would be illegal, because SALARY could not contain a value enabling the condition to be true. Similarly, you cannot compare a field of data type INTEGER to an expression whose value is not within the limits for values storable in INTEGER fields.
- You cannot compare a PIC field formatted without a negative sign to an expression whose value is negative. For example, if TOTAL were a PIC field without a sign, you could not use the condition P < 0, because it would necessarily be false.
- You cannot compare a character constant to a character field that it doesn't fit into. For example, if MONTH were CHAR (3), then the expression X = "JULY" would be invalid. You cannot compare wildcards or ranges.

Functions

A function accepts one or more arguments, performs an action on them, and returns a value based on the action. A function is considered an expression because it evaluates to a single value. For all functions, a space is optional between a function's name and any parentheses that enclose its arguments.

Function Syntax

function:

numeric_conversion_function

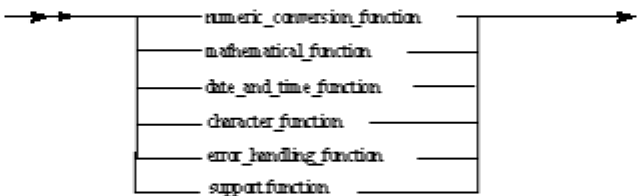
mathematical_function

date_time_function

character_function

error_handling_function

support_function



This section includes information about the following functions:

- Numeric Conversion Functions
- Mathematical Functions
- Date, Time and Timestamp Functions
- Character String Functions
- Double-Byte Character Set Functions
- Dynamically-Set View Functions in Java
- Error-Handling Functions
- Support Functions

For language and release-version specific considerations on functions, refer to Platform Support and Target Language Specifics.

Numeric Conversion Functions

The numeric conversion functions INT and DECIMAL convert character strings to numeric (integer or decimal) values.

The difference between the INT function and the DECIMAL function is the return type. The INT function returns INTEGER; the DECIMAL function

returns DEC of appropriate length and scale. The result of the INT function must be an integer not exceeding $2^{31} - 1$, otherwise the result is unpredictable. The result of the DECIMAL function must fit into the DEC Rules data type. For restrictions, see Restrictions on Features.

Numeric Conversion Function Syntax

numeric_conversion_function:

num_conv_func_name '(' character_data_item [, format_string] ')'

num_conv_func_name:

one of INT DECIMAL

See the following topics for more information and examples:

- INT and DECIMAL Functions with One Parameter
- INT and DECIMAL Functions with Two Parameters
- Numeric Conversion Functions: Troubleshooting

INT and DECIMAL Functions with One Parameter

A character string can contain some symbols for readability. The character string that is to be converted to a numeric value can contain the following symbols besides digits (0-9):

- leading and trailing spaces, which are ignored.
- plus and minus sign at the beginning can optionally be followed by spaces before the first digit.
- any number of locale-specific thousand separators in any position inside the number, which are ignored.
- one locale-specific decimal separator, for DECIMAL function only.
- minus sign at the end of the number itself, which makes this number negative as well as the leading minus sign; spaces before this minus sign are not allowed.
- one currency symbol in any position, which is ignored.

If the input string is empty (contains spaces only) or does not comply with the format described above, integer zero is returned.

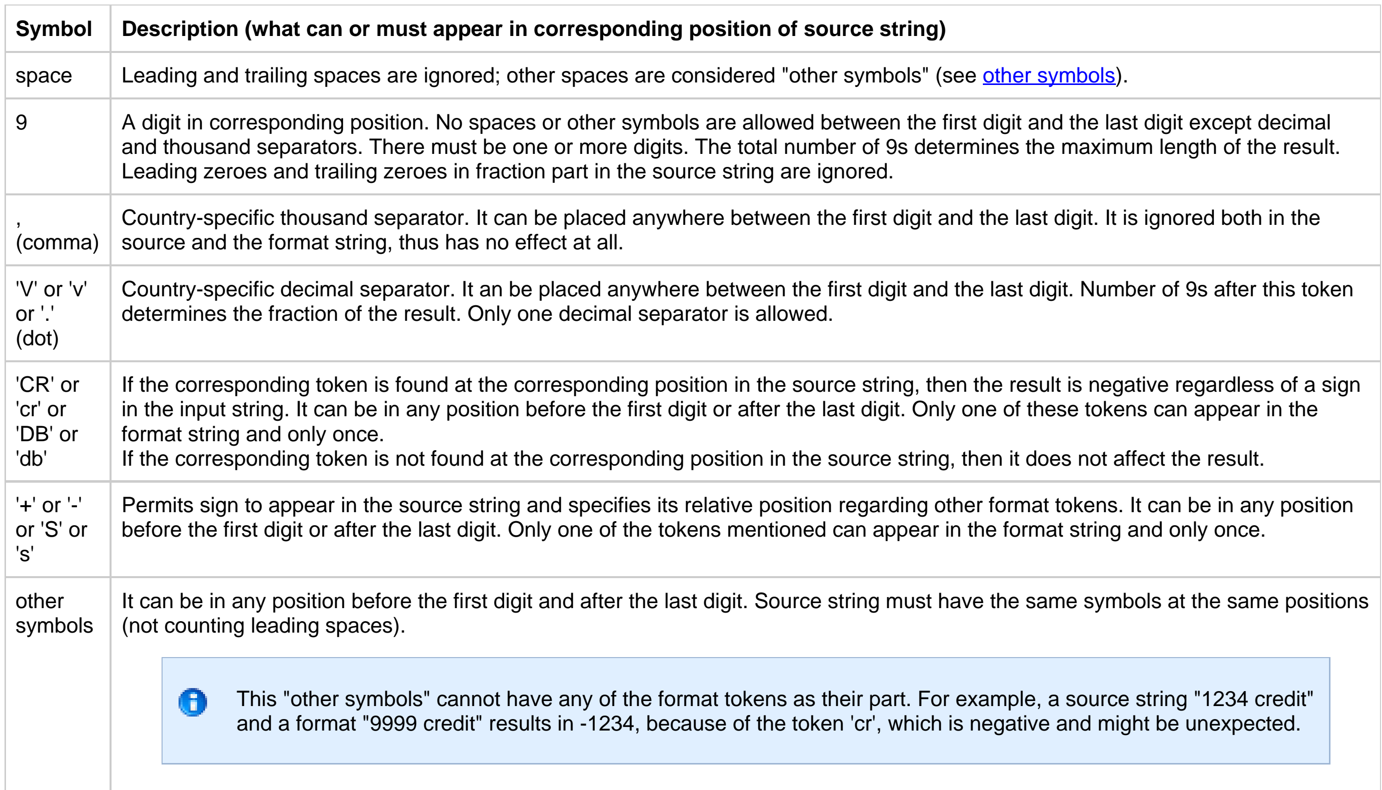
Examples: INT and DECIMAL functions with one parameter

```
DCL
   I INTEGER;
   D DEC(15, 5);
ENDDCL

MAP INT("123") TO I // I=123
MAP INT("$123,456") TO I // I=123456
MAP INT("  $123,456-   ") TO I // I=-123456
MAP INT("  +$123,456") TO I // I=123456
MAP INT("$123,456$") TO I // I = 0 because of two currency symbols
MAP INT("-123-456") TO I // I = 0 because of two minus signs
MAP INT("123456 -") TO I // I = 0 because of space after digit but before minus sign
MAP INT("123 credit") TO I // I = 0 because of invalid characters in the input string
SET I:= INT("123.12") // I = 0 because of decimal separator
MAP DECIMAL("123.4-") TO D // D=-123.4
MAP DECIMAL("+123.4$") TO D // D=123.4
MAP DECIMAL("123,456,7") TO D // D=1234567
```

INT and DECIMAL Functions with Two Parameters

When supplying the second parameter, *format_string*, follow the format string tokens as described in the table below.

Format string tokens



Symbol	Description (what can or must appear in corresponding position of source string)
space	Leading and trailing spaces are ignored; other spaces are considered "other symbols" (see other symbols).
9	A digit in corresponding position. No spaces or other symbols are allowed between the first digit and the last digit except decimal and thousand separators. There must be one or more digits. The total number of 9s determines the maximum length of the result. Leading zeroes and trailing zeroes in fraction part in the source string are ignored.
, (comma)	Country-specific thousand separator. It can be placed anywhere between the first digit and the last digit. It is ignored both in the source and the format string, thus has no effect at all.
'V' or 'v' or '.' (dot)	Country-specific decimal separator. It an be placed anywhere between the first digit and the last digit. Number of 9s after this token determines the fraction of the result. Only one decimal separator is allowed.
'CR' or 'cr' or 'DB' or 'db'	If the corresponding token is found at the corresponding position in the source string, then the result is negative regardless of a sign in the input string. It can be in any position before the first digit or after the last digit. Only one of these tokens can appear in the format string and only once. If the corresponding token is not found at the corresponding position in the source string, then it does not affect the result.
'+' or '-' or 'S' or 's'	Permits sign to appear in the source string and specifies its relative position regarding other format tokens. It can be in any position before the first digit or after the last digit. Only one of the tokens mentioned can appear in the format string and only once.
other symbols	It can be in any position before the first digit and after the last digit. Source string must have the same symbols at the same positions (not counting leading spaces). This "other symbols" cannot have any of the format tokens as their part. For example, a source string "1234 credit" and a format "9999 credit" results in -1234, because of the token 'cr', which is negative and might be unexpected.

The actual length and scale of the returned result are determined by the input string contents rather than the format string specifications. The format string restricts length and scale of the returned value but does not specify the exact length and scale. Both INT function and DECIMAL function accept the same input and format strings; however, the fraction part is discarded for INT function.

Country-specific settings are used in Java and C only.

List of Error Situations and Warnings

According to the format string specifications, the list of *error situations* is the following:

- Any format symbol, except 9 . V v . , between the first and the last digit symbol.
- More than one decimal separator.
- More than one sign symbol.
- More than one Credit/Debit symbol.
- More than one currency symbol.
- Wrong position of comma (used after decimal separator).

Warnings will be issued in the following situations:

- Format string is empty.
- Empty whole part.
- No digit symbols found.
- No digit signs found in a format string.

Example: INT and DECIMAL functions with two parameters

The following table describes the sample results when INT and DECIMAL functions are used with the format string (the second parameter).

```
DCL
   I INTEGER;
   D DEC(15, 5);
ENDDCL

MAP INT ("source_string", "format_string") TO I
MAP DECIMAL ("source_string", "format_string") TO D
```

Sample results of INT and DECIMAL functions with two parameters

Source String	Format String	Result	Comments
" 123 "	"999"	123	Leading and trailing spaces are ignored.
"12 3"	"999" or "99 9"	0	Spaces inside the number are not allowed.
"123,000"	"9,999,999"	123000	Thousand separators are ignored.
"123.456"	" 999v99"	123.45	Fraction part that does not fit into format is discarded.
"123.456"	"99.99"	0	Integer part is too large to fit into format.
" (123.4-)"	"(999v99s)"	-123.4	
"(123.4)"	"(s999v99)"	0	The source string does not comply with the format string because of spaces after the opening bracket.
"123"	"s999" or "999s"	123	It is not necessary for a sign to be present in the source string.
"-123"	"999s"	0	Sign in the source string is at the beginning, but the format string expects it at the end.
"-123cr" "123cr" "0cr"	"-999cr" "999cr" "999cr"	-123 -123 0	If 'cr' is specified, the return value is always negative (unless it is zero).
"0123"	"9999"	123	The return type becomes DEC(3,0) (size of DEC determined by the value).
"" "123"	"-999.9" ""	0 0	Empty string implies zero result regardless of the format string, and empty format string implies zero result regardless of the source string.
"123 dollars"	"999 dollars"	0	's' in format string implies sign, but there is no sign at the corresponding position in the input string, and 's' is skipped after this check. But this leaves 's' in the input string unmatched.
"123c"	"999crc"	123	'cr' in the format string is skipped and then 'c' matches the corresponding symbol in the input string.

Numeric Conversion Functions: Troubleshooting

The following table describes possible errors and descriptions associated with INT and DECIMAL functions:

Error situations and handling

Error Description	Error Handling and Returned Result
Input string contains more than 31 digits, but less or equal to SIZE_OF_DEC	If the integer part is longer than 31 digits, integer zero is returned. If the integer part length is not longer than 31 digits, but the total length exceeds 31, all digits in the fraction part that do not fit into 31 are ignored.
Either format or input string length exceeds SIZE_OF_DEC	Integer zero is returned. SIZE_OF_DEC is as follows: • C: 62 • ClassicCOBOL: 60 • OpenCOBOL: 32 • Java: 64

Any error in format string (for example, more than one decimal separator)	Integer zero is returned.
One or more input string symbols do not match corresponding format string symbol	Integer zero is returned.
Format string integer part is less than source string integer part (999.9 and 1234.5)	Integer zero is returned.
Format string fraction part is less than source string fraction part (999.9 and 123.45)	Digits that do not fit into format string are discarded, resulting 123.4.

Mathematical Functions

A mathematical function returns a value by manipulating the first parameter using the value specified in the second parameter.

Mathematical Function Syntax

mathematical_function:

math_func_for_numeric_expr

math_func_for_var_data_item

math_func_for_numeric_expr:

math_func_name1 ‘(‘ numeric_expression [, numeric_expression] ‘)’

math_func_name1:

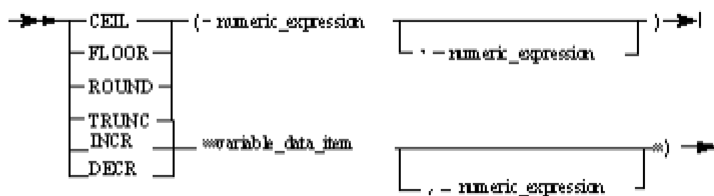
one of CEIL FLOOR ROUND TRUNC

math_func_for_var_data_item:

math_func_name2 ‘(‘ variable_data_item [, numeric_expression] ‘)’

math_func_name2:

one of INCR DECR



where:

- numeric_expression — see Numeric Expressions.
- variable_data_item is a variable data item of any numeric type.

When using CEIL, FLOOR, ROUND and TRUNC, the first expression is the value to be modified. The second expression specifies the significant number of digits to which the function applies — a positive value referring to digits to the right of the decimal point; zero referring to the digit to the immediate left of the decimal point; and a negative value referring to digits further to the left of the decimal point. For instance, a second expression of 2 refers to the hundredths place, and a second expression of -2 refers to the hundreds column to the left of the decimal point. An omitted second expression is the equivalent of 0 and applies the function to the nearest integer value. The data type of the returned value for any of these functions is DEC.

INCR and DECR modify their arguments, but the other mathematical functions do not. CEIL, FLOOR, ROUND and TRUNC only return a value calculated based on the arguments.

See the following for more information about each of the mathematical functions and examples:

- CEIL
- FLOOR
- ROUND
- TRUNC
- INCR
- DECR

CEIL

The CEIL function, which is short for ceiling, returns the next number greater than the first expression to the significant number of digits indicated by the second expression.

CEIL Examples

```
MAP CEIL (1234.5678, 2)   TO X  *> sets X to 1234.57 <*
MAP CEIL (1234.5678, 1)   TO X  *> sets X to 1234.6  <*
MAP CEIL (1234.5678)      TO X  *> sets X to 1235    <*
MAP CEIL (1234.5678, -1)  TO X  *> sets X to 1240    <*
MAP CEIL (1234.5678, -2)  TO X  *> sets X to 1300    <*
MAP CEIL (-1234.5678, 1)  TO X  *> sets X to -1234.5 <*
MAP CEIL (-1234.5678)     TO X  *> sets X to -1234   <*
MAP CEIL (-1234.5678, -1) TO X  *> sets X to -1230   <*
```

FLOOR

This function returns the next number less than the first expression to the significant number of digits indicated by the second expression.

FLOOR Examples

```
MAP FLOOR (1234.5678, 2)   TO X  *> sets X to 1234.56 <*
MAP FLOOR (1234.5678, 1)   TO X  *> sets X to 1234.5  <*
MAP FLOOR (1234.5678)      TO X  *> sets X to 1234    <*
MAP FLOOR (1234.5678, -1)  TO X  *> sets X to 1230    <*
MAP FLOOR (1234.5678, -2)  TO X  *> sets X to 1200    <*
MAP FLOOR (-1234.5678, 1)  TO X  *> sets X to -1234.6 <*
MAP FLOOR (-1234.5678)     TO X  *> sets X to -1235   <*
MAP FLOOR (-1234.5678, -1) TO X  *> sets X to -1240   <*
```

ROUND

The ROUND function returns the number closest to the first expression to the significant number of digits indicated by the second expression. It sets any rounded digits to zero. The ROUND function observes the usual rounding conventions (0 – 4 for rounding down, 5 – 9 for rounding up). The second expression must always be greater than or equal to -10 or less than or equal to 10 (-10 x 10).

ROUND Examples

```
MAP ROUND (1234.5678, 2)   TO X  *> sets X to 1234.57 <*
MAP ROUND (1234.5678, 1)   TO X  *> sets X to 1234.6  <*
MAP ROUND (1234.5678)      TO X  *> sets X to 1235    <*
MAP ROUND (1234.5678, -1)  TO X  *> sets X to 1230    <*
MAP ROUND (1234.5678, -2)  TO X  *> sets X to 1200    <*
MAP ROUND (-1234.5678, 1)  TO X  *> sets X to -1234.6 <*
MAP ROUND (-1234.5678)     TO X  *> sets X to -1235   <*
MAP ROUND (-1234.5678, -1) TO X  *> sets X to -1230   <*
```

TRUNC

The TRUNC function, which is short for truncate, returns a number that is the first expression with any digits to the right of the indicated significant

digit set to zero. It removes a specified number of digits from the first expression.

TRUNC Examples

```
MAP TRUNC (1234.5678, 2)   TO X  *> sets X to 1234.56 <*
MAP TRUNC (1234.5678, 1)   TO X  *> sets X to 1234.5  <*
MAP TRUNC (1234.5678)      TO X  *> sets X to 1234    <*
MAP TRUNC (1234.5678, -1)  TO X  *> sets X to 1230    <*
MAP TRUNC (1234.5678, -2)  TO X  *> sets X to 1200    <*
MAP TRUNC (-1234.5678, 1)  TO X  *> sets X to -1234.5 <*
MAP TRUNC (-1234.5678)     TO X  *> sets X to -1234   <*
MAP TRUNC (-1234.5678, -1) TO X  *> sets X to -1230   <*
```

INCR

The INCR function, which is short for INCRement, adds its second parameter to its first parameter and returns the modified first parameter. If the second parameter is omitted, INCR adds 1 to the first parameter and returns the modified first parameter. This function can be used in an expression and as a statement.

Because the INCR function modifies its left operand and the order of calculation is different on different platforms, the same expression might give different results on different platforms. See INCR and DECR in Java and INCR and DECR in OpenCOBOL for examples. Use this function in expressions with caution.

If the type of the second parameter differs from the type of the variable (the first parameter), then the same conversions apply as for "MAP param1 + param2 TO param1", where param1 and param2 are the first and the second parameters of the INCR function respectively.

INCR Examples

```
MAP INCR(1)+1 TO I  *> Wrong - first parameter must be a variable <*

MAP INCR(I+1) TO I  *> Wrong - first parameter must be a variable <*


MAP 1 TO I
MAP INCR(I) TO J    *> sets I and J to 2 <*
MAP INCR(I,2) TO J  *> sets I and J to 4 <*
INCR(I,J)           *> sets I to 8, J is left unchanged <*
INCR(I,J+1)         *> sets I to 13, J is left unchanged <*

MAP 1 TO I,J
MAP J+INCR(I) TO J  *> sets I to 2 and J to 3 <*
```

DECR

The DECR function, which is short for DECRement, subtracts its second parameter from its first parameter and returns the modified first parameter. If the second parameter is omitted, DECR subtracts 1 from the first parameter and returns the modified first parameter. This function can be used in an expression and as a statement.

Because the DECR function modifies its left operand and the order of calculation is different on different platforms, the same expression might give different results on different platforms. See INCR and DECR in Java and INCR and DECR in OpenCOBOL for examples. Use this function in expressions with caution.

If the type of the second parameter differs from the type of the variable (the first parameter), then the same conversions apply as for "MAP param1 - param2 TO param1", where param1 and param2 are the first and the second parameters of the DECR function respectively.

DECR Examples

```
MAP DECR(1)+1 TO I  *> Wrong - first parameter must be a variable <*

MAP DECR(I+1) TO I  *> Wrong - first parameter must be a variable <*


MAP 13 TO I
MAP DECR(I) TO J    *> sets I and J to 12 <*
MAP DECR(I,2) TO J  *> sets I and J to 10 <*
DECR(I,J)           *> sets I to 0, J is left unchanged <*
DECR(I,J+1)         *> sets I to -11, J is left unchanged <*

MAP 1 TO I,J
MAP J+DECR(I) TO J  *> sets I to 0 and J to 1 <*
```

C generation restrictions

Semantic of arithmetic operation calculation and passing parameters to functions is platform dependent in Rules Language.

You should be very careful when using:

- expressions with side effect in order to pass parameters to any function
- expressions with side effect in complex expressions

There are two types of side effect expressions in Rules Language:

1. *Local procedure* that changes global data of a rule – see Example 1.
2. *INCR* and *DECR* standard functions of Rules Language.

Example 1

```
dcl
   i integer;
   j integer;
enddcl

proc func_with_side_effect(arg integer) : integer
   map arg to i
   proc return i
endproc

map 0 to i
map i + func_with_side_effect(1) to j
```

Since a function with side effect is used in the same expression with data that this function affects, the result of *i + func_with_side_effect(1)* might not be the same for all generation types. In this case, *j* may have value *1* or *2*.

Workaround

Try not to use a function that changes some global data, especially when global data itself is the range of one expression.

The code shown in Example 2 and Example 3 gives the same result for all platforms (use one that is correct for your application):

Example 2

```
map 0 to i
map func_with_side_effect(1) to j
map i + j to j
// (j = 2)
```

or

```
map 0 to i
map i to j
map func_with_side_effect(1) + j to j
// (j = 1)
```

Example 3

```
dcl
   i integer;
   j integer;
enddcl

map 0 to i
map decr(i) + incr(i) to j

// functions with side effect ICNR and DECR operate with the same data. It is not guarantied that the
result is the same for different platforms.
```

Workaround

Try to use INCR and DECR functions as standalone statements where it is possible.

When the result of INCR or DECR is used in an expression, make sure that its operand is not used anywhere else in that expression. Example 4 illustrates the correct code:

Example 4

```
map incr(i) + incr(j) to j

// i and j are used once. It is correct.
```

It is also not recommended to pass expressions with side effect (INCR, DECR or a local procedure that changes global data) to the procedure parameters (see Example 5).

Example 5

```
map 0 to i
map some_prec(i, incr(i))

//actual parameters passed to the procedure can be (1, 1) or (0, 1), depending on platform specific
parameter passing rules.
```

Workaround

It is not recommended to use a function that changes some global data and global data itself in ranges on one local procedure call. See Example 6 for a correct code sample:

Example 6

```
map 0 to i
map i to temp
map some_prec(temp, incr(i))
```

or

```
map 0 to i
map incr(i) to temp
map some_prec(i, temp)

//depending on the application logic.
```

Date, Time and Timestamp Functions

Use a DATE, TIME and TIMESTAMP function to obtain the current date, time and timestamp, to format your data, or to convert a field from a date, time or timestamp data type to another data type.

The system date, time, and timestamp are unique to each workstation. You (or your system administrator) must synchronize your workstations if you want the values to be consistent. Alternatively, you can run any rules that call for a date, time, or timestamp on the host.

Date, Time and Timestamp Functions Syntax

date_time_function:

    date_time_timestamp_functions_without_parameters

    date_function

    time_function

    timestamp_function

    integer_conversion_function

    character_conversion_function

    TIMESTAMP ‘(‘ *date_field*, *time_field*, *fraction* ‘)’

date_time_timestamp_functions_without_parameters:

    one of DATE TIME TIMESTAMP

date_function:

    date_func_with_one_parameter

    date_func_with_two_parameters

date_func_with_one_parameter

    date_func_name1 ‘(‘ *date_field* ‘)’

date_func_name1:

    one of DAY MONTH YEAR DAY_OF_YEAR DAY_OF_WEEK CHAR INT NEW_TO_OLD_DATE ISLEAPYEAR

date_func_with_two_parameters:

    CHAR ‘(‘ *date_field*, *format_string* ‘)’

time_function:

    time_func_with_one_parameter

    time_func_with_two_parameters

time_func_with_one_parameter:

time_func_name1 ‘(‘ *time_field* ‘)’

time_func_name1:

one of SECONDS MILSECS MINUTES HOURS SECONDS_OF_DAY MINUTES_OF_DAY CHAR INT NEW_TO_OLD_TIME

time_func_with_two_parameters:

CHAR ‘(‘ *time_field*, *format_string* ‘)’

timestamp_function:

timestamp_func_with_one_parameter

timestamp_func_with_two_parameters

timestamp_func_with_one_parameter

timestamp_func_name1 ‘(‘ *timestamp_field* ‘)’

timestamp_func_name1:

one of DATE TIME FRACTION CHAR

timestamp_func_with_two_parameters:

CHAR ‘(‘ *timestamp_field*, *format_string* ‘)’

integer_conversion_function:

integer_conversion_func_name1 ‘(‘ *integer_field* ‘)’

integer_conversion_func_name1:

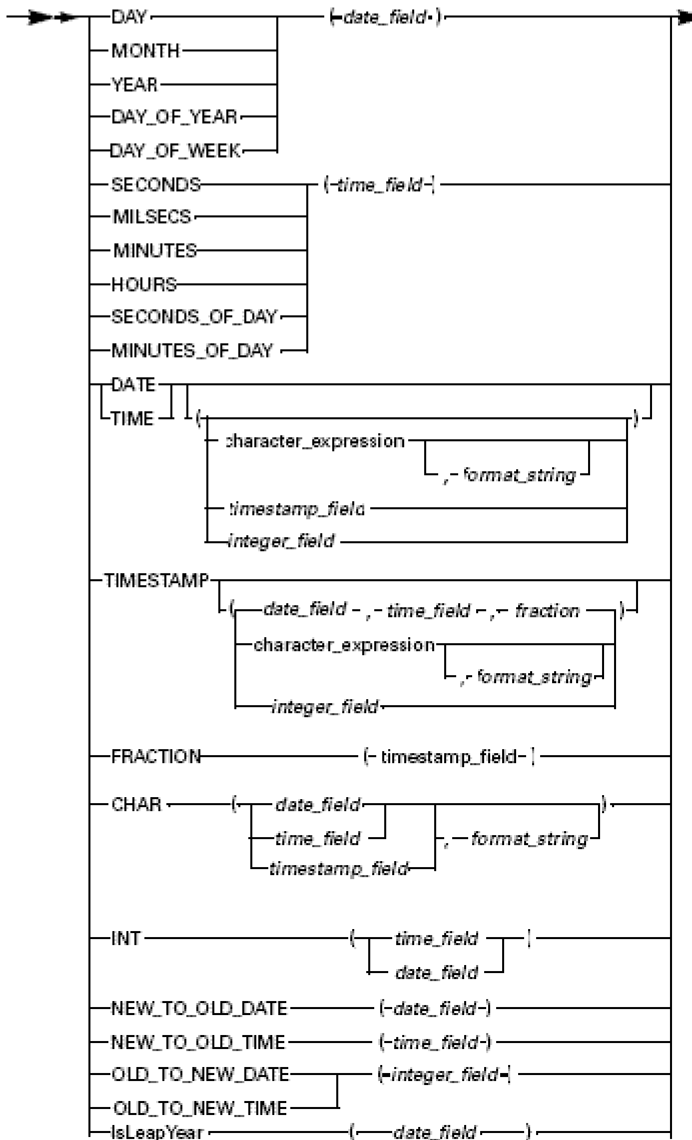
one of DATE TIME TIMESTAMP OLD_TO_NEW_DATE OLD_TO_NEW_TIME

character_conversion_function:

character_conversion_func_name1 ‘(‘ character_expression [, *format_string*] ‘)’

character_conversion_func_name1:

one of DATE TIME TIMESTAMP



where:

- character_expression — see Character Expressions.
- *date_field* — see Date and Time Data Types.
- *time_field* — see Date and Time Data Types.
- *integer_field* — see Numeric Data Types.
- *format_string* — see Format String.
- *timestamp_field* — see Date and Time Data Types.

See these related topics for detailed information:

- Date and Time Function Definitions
- Format String
- Common Separators
- Date Format String
- Time Format String
- Timestamp Format String
- Input String Restrictions
- Sample Date, Time and Timestamp Functions

Date and Time Function Definitions

The following table describes the Date and Time functions:

Date and Time Functions

Function Name	Determines...	Returns...
DAY	The date that the value in the specified date field represents.	The day of the month for that date in a SMALLINT field.
MONTH	The date that the value in the specified date field represents.	The month of the year for that date in a SMALLINT field.
YEAR	The date that the value in the specified date field represents.	The year for that date in a SMALLINT or INTEGER field.
DAY_OF_YEAR	The date that he value in the specified date field represents.	The Julian day of the year for that date in a SMALLINT field.
DAY_OF_WEEK	The date that the value in the specified date field represents.	The day of the week for that date in a SMALLINT field.
SECONDS	The date that the time in the specified date field represents.	The number of seconds past the minute for that time in a SMALLINT field.
MILSECS	The time that the value in the specified time field represents.	The number of milliseconds past the second for that time in a SMALLINT field.
MINUTES	The time that the value in the specified time field represents.	The number of minutes past the hour for that time in a SMALLINT field.
HOURS	The time the value in the specified time field represents.	The number of hours since midnight for that time in a SMALLINT field.
SECONDS_OF_DAY	The time the value in the specified time field represents.	The number of seconds since midnight for that time in an INTEGER field.
MINUTES_OF_DAY	The time the value in the specified time field represents	The number of minutes since midnight for that time in a SMALLINT field.
DATE **and** **TIME** You can use this function with or without an argument	Without an argument	DATE returns the current system date in a DATE field, and TIME returns the current system time in a TIME field.
	With a value in a TIMESTAMP field as an argument, it determines the value that the date or time portion of that field represents.	DATE returns the date in a DATE field, and TIME returns the time in a TIME field.
	With a character value as an argument, these functions convert the character value to a date or a time, in a DATE field. It interprets the character value according to a format string. See Format String for tokens to use in a format string. If you omit the format string, the default format strings are provided. See the following for language specific considerations: • Date and Time Functions in C • Date and Time Functions in Java • Date and Time Functions in OpenCOBOL DATE and TIME with DBCS and MIXED argument is available in Java and COBOL for DBCS enabled releases. See Restrictions on Features. \\\\	DATE returns the date in a DATE field, and TIME returns the time in a TIME field.
	With an integer value as an argument, these functions convert the integer value to a date or a time.	DATE returns the date in a DATE field, and TIME returns the time in a TIME field.

TIMESTAMP You can use this function with or without arguments.	Without arguments	A timestamp created from the current date and time in a TIMESTAMP field.
	With arguments, it concatenates the three fields.	The value in a TIMESTAMP field.
	With a character value as an argument, this function converts the character value to a timestamp, in a TIMESTAMP field. It interprets the character value according to a format string. See Format String for tokens to use in a format string. TIMESTAMP is available in OpenCOBOL, Java, and CSharp.	TIMESTAMP returns the timestamp in a TIMESTAMP field.
	With an integer value as an argument, this function converts the integer value to a timestamp value.	TIMESTAMP returns the timestamp in a TIMESTAMP field.
FRACTION	The timestamp represented by the value in the specified timestamp field. See FRACTION in OpenCOBOL for more information.	The number of picoseconds for that timestamp in an INTEGER field. Maximum value is 999,999,999 which represents the fraction of a millisecond. The entire number of milliseconds is included into the TIME component of the TIMESTAMP.
CHAR	Converts a value in a DATE, TIME or TIMESTAMP field to a value in a CHAR field. It formats the character value according to the system default unless you provide a format string. See Format String for tokens to use in a format string. The TIMESTAMP argument of the function is available only in OpenCOBOL.	The returned value is formatted as specified in the Data Types data types.
INT	Converts the time or date in the specified time or date field. If the DATE/TIME value is invalid, -1 is returned. See Date and Time Functions in OpenCOBOL for exception.	A value in an INTEGER field.
NEW_TO_OLD_DATE	Converts the date in the specified DATE field.	A value in an INTEGER field. The INTEGER field represents date in the old format. Date in the old format is integer, where the 4 first decimal digits represent year, the other 2 digits stand for the month, and the last 2 digits indicate the day.
NEW_TO_OLD_TIME	Converts the time with the specified TIME field.	A value in an INTEGER field. The format of the return value is HHMMSS in all generations except ClassicCOBOL and OpenCOBOL for which it depends on parameter OLD_TO_NEW_TIME_MODE (see ClassicCOBOL specific settings in CodegenParameters section and OpenCOBOL specific settings in CodegenParameters section).
OLD_TO_NEW_DATE	Converts the value in the specified INTEGER field. The INTEGER field represents date in the old format. Date in the old format is integer, where the 4 first decimal digits represent year, the other 2 digits stand for the month, and the last 2 digits indicate the day.	A value in a DATE field.
OLD_TO_NEW_TIME	Converts the value with the specified INTEGER field.	A value in a TIME field. This function accepts values in format HHMMSS in all generations except ClassicCOBOL and OpenCOBOL for which it depends on parameter OLD_TO_NEW_TIME_MODE (see ClassicCOBOL specific settings in CodegenParameters section and OpenCOBOL specific settings in CodegenParameters section).
IsLeapYear	With a DATE value as an argument, this function determines whether the year is leap or not.	IsLeapYear returns a BOOLEAN value.

All the date and time functions that deal with years assume by default a twentieth (20th) century when they encounter two-digit years in their parameter (so, by default, all two-digit years are preceded by 19). You can control this behavior by setting the Hps.ini file [AE Runtime] section DEFAULT_CENTURY key. If this key is set to "2000", all date and time functions assume twenty-first (21st) century when dealing with two-digit years (that is, all two-digit years are preceded by 20).

For DATE function, if both %c and %Y tokens are specified then the first two digits of the value of the year token are ignored. For example, both DATE('09/12/20/1945','%m/%d/%c/%Y') and DATE('09/12/1945/20','%m/%d/%Y/%c') return the date in which the value of the year is 2045.

The %f and %0f Timestamp format tokens do not represent the entire FRACTION field. They represent milliseconds from the TIME field and microseconds from the FRACTION field of the TIMESTAMP value

Format String

For a DATE, TIME or TIMESTAMP function, use a format string to tell the system how to interpret a character value when converting it to a value in a date or time field. For a CHAR function, use a format string to format a value in a date or time field when converting it to a character field. If you do not provide a format string, the default system format as set during installation is used. See Date and Time Data Types for the default format.

A format string consists of a series of tokens enclosed in single quotation marks. You can provide these tokens either as a literal or in a character variable. Provide one token for each element of the date or time. For a date value, for instance, provide one token for day, one for month, and one for year. Separate the tokens with the same separators used in the provided value. If you do not use a separator, any value stored in a date or time field might be ambiguous.

Date Format Tokens, Time Format Tokens and Timestamp Format Tokens list the tokens you can place into a format string. The separators are common to DATE, TIME and TIMESTAMP fields, but the other tokens are not.

The following limitations also apply to these functions:

- The returned value depends on the current NLS settings.
- A format string is not interpreted until runtime. This means that a format string is not validated during preparation and a statement with an incorrect format string prepares successfully.
- If a function cannot convert an input string, a numeric function returns a value of -1 and a character function returns the null string.
- A format string is case-sensitive.
- If only one argument is specified, that argument is considered to be the input string, not the format string. Therefore, if an incorrect template is specified as the only argument – (for example, DATE ('m%f%) – the rule prepares successfully. This is because the format string 'm%f%y' is a valid literal string. If you do not provide a format string, the template provided in the language configuration file is used. See Date and Time Data Types for more information.
- The %x time token is ignored with either %0t, %t or %H. The AM/PM flag can be specified in the following forms:

  ```
  AM/PM, A M/P M, A.M./P.M.
  ```

- -1 is returned as a result where a format string contains two tokens in sequence that are not delimited by a separator and first of the tokens is

  %m, %d, %y, %c, %j for DATE function

  or

  %h, %t, %m, %s, %f for TIME function.

  This occurs because these tokens accept an unlimited number of digits.

  Any symbol can be used as a delimiter. For example:

  Procedure

  `TIME ('1 25', '%h %m')` returns 1:25

  `DATE ('12.01.1998', '%d.%m.%c%0y')` returns -1 because 1998 is considered century and %c accepts an unlimited number of digits, so there is no value for the year.

  `TIME ('125', '%h%m')` is ambiguous and can be interpreted as 12:05 or 1:25 because the tokens are defined as
  - %h = Hour, numeric (0..12)
  - %m = Minute, numeric (0..59)

This restriction currently applies only to DATE and TIME functions; it is not enforced for the CHAR function.

- One token must be specified for year, month, and day in the DATE function and one token must be specified for hours and minutes in the TIME function. Otherwise, -1 is returned as a result.

  The exception to this rule appears when the Julian specifier is submitted. A Julian date value inherently specifies month and day, therefore only a Year should be required.

If the century token is specified and the year token is not, the result is the first year of that century.

- `DATE ('12/1998', '%m/%Y')` is invalid because the value for the day is missing.
- `TIME (':25', ':%m')` is invalid because the value for the hour is missing.
- `TIME (":25", '%h:%m')` is invalid because %h means there should be at least one digit in the hour value (0..12).
- If more than one token of the same type is specified (%y and %Y or %D and %d) in the DATE or TIME functions, then an error with the result of -1 is returned.
- `DATE ('12/23/01/1998', '%m/%d/%0d/%Y')` is invalid because the value of the day token is ambiguous; it can be either 1 or 23
- The AM/PM flag can be specified in the TIME function in the form:

  ```
  AM/PM, A M/P M, A.M./P.M.
  ```

- For TIMESTAMP function, there are 3 different tokens for month: %o, %0o, and %O. These tokens replace %m, %0m, and %M, used for minutes in a TIMESTAMP function.

See also Date and Time Functions in OpenCOBOL.

For locale-specific format tokens %M, %W, %D and %O, the length of the resulting string cannot be accurately predicted at the time of rule preparation; so it is assumed to be 256 symbols. If the real length of a destination string is less than 256, a warning is generated that the result may be truncated.

Common Separators

Use the separators in the following table for both date and time values.

Format String Separators

-
/
:
,
.
;

Date Format String

Use the tokens in the following table when formatting a DATE value.

Date Format Tokens

Token	Description	Example
%m	Month, numeric (1...12)	2
%0m	Month, numeric, with leading zero (01...12)	02
%M	Month, word (January...December)	February
%d	Day, numeric (1...31)	28
%0d	Day, numeric, with leading zero (01...31)	28

%D	Day, ordinal (1st...31st)	28th
%j	Day, Julian (1...366)	59 (Feb. 28)
%0j	Day, Julian, with leading zero (01...366)	59 (Feb. 28)
%c	Year, numeric, first 2 digits (century minus 1)	19
%0c	Year, numeric, first 2 digits with leading zero (century minus 1)	19
%y	Year, numeric, last 2 digits (00...99), with the first two digits implied to be 19[1]	95
%0y	Year, numeric, last 2 digits, with leading zero (00...99)	95
%Y	Year, numeric, all 4 digits (0000...9999)	1995
%W	Weekday, word (Sunday...Saturday)	Tuesday

1. If the last digits of the year are in the following range: 00 - 09, the CHAR function truncates the leading zero in part of the result corresponding to "%y" token. "%y" token in the format string of DATE function accepts any number of digits and uses the first two of them.

Time Format String

Use the tokens in the following table when formatting a TIME value.

Time Format Tokens

Token	Description	Example
%h	Hour, numeric (1...12)	2
%0h	Hour, numeric, with leading zero (01...12)	02
%t	Hour, numeric (0...23)	14
%0t	Hour, numeric, with leading zero (00...23)	14
%T	Hour, word (one...twelve)	Eleven
%H	Hour, word (zero...twenty-three)	Fourteen
%m	Minute, numeric (0...59)	45
%0m	Minute, numeric, with leading zero (00...59)	45
%M	Minute, word, (zero...fifty-nine)	Forty-five
%s	Second, numeric (0...59)	9
%0s	Second, numeric, with leading zero (00...59)	09
%S	Second, word (zero...fifty-nine)	Nine
%f	Millisecond, numeric (0...999)	89
%0f	Millisecond, numeric, with leading zeroes (000...999)	089
%x	Ante (AM) or post (PM) meridiem	PM

Timestamp Format String

Use this tokens in the following table when formatting a TIMESTAMP value.

Timestamp Format Tokens

Token	Description	Example
%o	Month, numeric (1...12)	2
%0o	Month, numeric, with leading zero (01...12)	02
%O	Month, word (January...December)	February

%d	Day, numeric (1...31)	28
%0d	Day, numeric, with leading zero (01...31)	28
%D	Day, ordinal (1st...31st)	28th
%j	Day, Julian (1...366)	59 (Feb. 28)
%0j	Day, Julian, with leading zero (01...366)	59 (Feb. 28)
%c	Year, numeric, first 2 digits (century minus 1)	19
%0c	Year, numeric, first 2 digits with leading zero (century minus 1)	19
%y	Year, numeric, last 2 digits (00...99), with the first two digits implied to be 19[2]	95
%0y	Year, numeric, last 2 digits, with leading zero (00...99)	95
%Y	Year, numeric, all 4 digits (0000...9999)	1995
%W	Weekday, word (Sunday...Saturday)	Tuesday
%h	Hour, numeric (1...12)	2
%0h	Hour, numeric, with leading zero (01...12)	02
%t	Hour, numeric (0...23)	14
%0t	Hour, numeric, with leading zero (00...23)	14
%T	Hour, word (one...twelve)	Eleven
%H	Hour, word (zero...twenty-three)	Fourteen
%m	Minute, numeric (0...59)	45
%0m	Minute, numeric, with leading zero (00...59)	45
%M	Minute, word, (zero...fifty-nine)	Forty-five
%s	Second, numeric (0...59)	9
%0s	Second, numeric, with leading zero (00...59)	09
%S	Second, word (zero...fifty-nine)	Nine
%f	Microsecond, numeric (0...999999)	8934
%0f	Microsecond, numeric, with leading zeroes (000...999999)	008934
%x	Ante (AM) or post (PM) meridiem	PM

2. If the last digits of the year are in the following range: 00 - 09, the CHAR function truncates the leading zero in part of the result corresponding to "%y" token. "%y" token in the format string of DATE function accepts any number of digits and uses the first two of them.

Input String Restrictions

Some restrictions apply for input strings when using the DATE, TIME and TIMESTAMP functions:

- Input strings must comply to the format string.

  `DATE ('12/1998', '%m/%d/%Y')` is invalid because the value of the day token is missing.
  `DATE ('12/1 1998', '%m/%d/%Y')` is invalid because the wrong delimiter is in the input string.
  `DATE ('12/1/98', '%m/%d/%Y')` is invalid because the year should contain four digits.

- All token values must be valid for corresponding format string elements. For more information, see Date Format Tokens, Time Format Tokens and Timestamp Format Tokens.

  `DATE ('12/0/1998', '%m/%d/%Y')` is invalid because the value of the day token is invalid.

- If %W token is specified in the DATE function, weekday must correspond to the date being specified by all the other tokens. Otherwise -1 is returned as a result. For example:

  `DATE ('12/28/1961 Saturday', '%m/%d/%Y %W')` is invalid because December 28, 1961 was Thursday.

- For TIMESTAMP function, there are 3 *different tokens for month*: %o, %0o, and %O. These tokens replace %m, %0m, and %M, used for

minutes in a TIMESTAMP function.

Sample Date, Time and Timestamp Functions

The following Rules code illustrates the use of most of the date and time functions. It assumes a country specification for the United States and assumes that the current system date and time are 7:26:03 P.M., January 26, 1995.

```
DCL
   DATE_VAR DATE;
   TIME_VAR TIME;
   TIMESTAMP_VAR TIMESTAMP;
   FRACTION_VAR INTEGER;
   INT_VAR INTEGER;
   SMALL_INT_VAR SMALLINT;
   CHAR_VAR CHAR (30);
ENDDCL

MAP DATE ('05/03/99', '%0m/%0d/%0y') TO DATE_VAR
MAP DATE ('5-3-99', '%m-%d-%y') TO DATE_VAR
MAP DATE ('Monday, May 3rd, 1999', '%W, %M %D, %Y') TO DATE_VAR
MAP DATE ('123;99', '%j;%y') TO DATE_VAR
*> All of these are equivalent; they place the value <*
*> for May, 3 1999 into DATE_VAR. <*

MAP TIME ('1:22:03 PM', '%h:%0m:%0s %x') TO TIME_VAR
MAP TIME ('13/22/3', '%t/%m/%s') TO TIME_VAR
MAP TIME ('One twenty-two three PM', '%T %M %S %x') TO TIME_VAR
*> All of these are equivalent; they place the value for <*
*> 1:22:03 PM into TIME_VAR. <*

MAP 0 TO FRACTION_VAR
*> Places the value 0 into FRACTION_VAR. <*

MAP TIMESTAMP (DATE_VAR, TIME_VAR, FRACTION_VAR) TO TIMESTAMP_VAR
*> Places the value for May 3, 1999 into the date portion of <*
*> the TIMESTAMP_VAR, the value for 1:22:03 PM into the time <*
*> portion, and the value 0 into the fraction portion. <*

MAP DATE TO DATE_VAR
*> Places the value for the system date, January 26, 1995, <*
*> into DATE_VAR. <*

MAP TIME TO TIME_VAR
*> Places the value for the system time, 7:26:03 PM, <*
*> into TIME_VAR. <*

MAP TIMESTAMP TO TIMESTAMP_VAR
*> Places the value for system timestamp into TIMESTAMP_VAR. <*

MAP CHAR (DATE_VAR, '%0m--%0d--%y') TO CHAR_VAR
*> Places the value '01--26--95' into CHAR_VAR. <*

MAP CHAR (DATE_VAR, '%M/%d, %c%y') TO CHAR_VAR
*> Places the value 'January/26, 1995' into CHAR_VAR. <*

MAP CHAR (TIME_VAR, '%H') TO CHAR_VAR
*> Places the value 'Nineteen' into CHAR_VAR. <*

MAP DAY (DATE_VAR) TO SMALL_INT_VAR
*> Places the value 26 into SMALL_INT_VAR. <*

MAP MONTH (DATE_VAR) TO SMALL_INT_VAR
*> Places the value 1 into SMALL_INT_VAR. <*

MAP YEAR (DATE_VAR) TO SMALL_INT_VAR
*> Places the value 1995 into SMALL_INT_VAR. <*
```

```
MAP DAY_OF_YEAR (DATE_VAR) TO SMALL_INT_VAR
*> Places the value 26 into SMALL_INT_VAR. <*

MAP SECONDS (TIME_VAR) TO INT_VAR
*> Places the value 3 into INT_VAR. <*

MAP MINUTES (TIME_VAR) TO INT_VAR
*> Places the value 26 into INT_VAR. <*

MAP HOURS (TIME_VAR) TO INT_VAR
*> Places the value 19 into INT_VAR. <*

MAP SECONDS_OF_DAY (TIME_VAR) TO INT_VAR
*> Places the value 69963 into INT_VAR. <*

MAP MINUTES_OF_DAY (TIME_VAR) TO INT_VAR
*> Places the value 1166 into INT_VAR. <*
```

```
MAP INT (DATE_VAR) TO INT_VAR
*>Places 728685 (number of days since Jan 1, 0001) into INT_VAR<*
```

The following Rules code illustrates the use of time, date and timestamp functions.

```
dcl
   ts TIMESTAMP;
   dt DATE;
   tm TIME;
   vc,vc1,vc2,vc3,vc4 VARCHAR(120);
enddcl

TRACE("Start")

map TIME("11.11.11.111", "%0t.%0m.%0s.%f") to tm
TRACE(tm)

map TIME("12.12.12.2", "%0t.%0m.%0s.%f") to tm
TRACE(tm)

map TIME("13.13.13.0", "%0t.%0m.%0s.%f") to tm
TRACE(tm)
TRACE('%f: ', CHAR(tm, "%0t.%0m.%0s.%f"))
TRACE('%0f: ', CHAR(tm, "%0t.%0m.%0s.%0f"))

map TIMESTAMP("00-00-0000.00.00.00.000000", "%0o-%0d-%Y.%0t.%0m.%0s.%f") to ts
TRACE(ts)

map TIMESTAMP("00-00-0000.00.00.00.000000", "%0o-%0d-%Y.%0t.%0m.%0s.%0f") to ts
TRACE(ts)

map TIMESTAMP("12-14-2005.12.13.14.445678") to ts
TRACE(ts)

map TIMESTAMP("12-04-2005.12.23.45.550000", "%0o-%0d-%Y.%0t.%0m.%0s.%0f") to ts
TRACE(ts)

map TIMESTAMP("12-04-2005.11 A M 23.45.111111", "%0o-%0d-%Y.%0h %x %0m.%0s.%f") to ts
TRACE(ts)

map CHAR(ts, '%0o-%0d-%Y.%0h %x %0m.%0s.%f') to vc4
TRACE('%f: ', vc4)
TRACE('%f: ', CHAR(ts, '%0o-%0d-%Y.%0h %x %0m.%0s.%f'))
TRACE('%0f: ', CHAR(ts, '%0o-%0d-%Y.%0h %x %0m.%0s.%0f'))

map TIME("21.15.38", "%0m.%0t.%0s") to tm
TRACE(tm)

map TIME("21.15.38", "%0m.%0t.%s") to tm
TRACE(tm)

map timestamp ('03-20-1977.15.21.38.321678', '%0o-%0d-%Y.%0t.%0m.%0s.%0f') to ts
TRACE(ts)

map timestamp ('03-20-1977.15.21.38.3216', '%0o-%0d-%Y.%0t.%0m.%0s.%f') to ts
TRACE(ts)

map CHAR(TIME, "hours: %h minutes %m second: %s") to vc
map CHAR(ts, "day: %d month: %o year: %Y hour: %h appm: %x minutes: %m sec: %s ms: %f Julian: %j
century: %c") to vc1
map CHAR(tm, "long string: hour: %h, minutes: %m, seconds: %s, milliseconds: %f") to vc2
map CHAR(ts, "O: %O D: %D W: %W H: %H T: %T S: %S") to vc3

map CHAR(ts) to vc

TRACE(ts)
TRACE(vc)
//TRACE(vc1)
//TRACE(vc2)
//TRACE(vc3)
```

The result that the code returns is the following:

```
Start
11:11:11:111
12:12:12:002
13:13:13:000
%f: 13.13.13.0
%0f: 13.13.13.000
**.*...*****
**.*...*****
2005-12-14.12.13.14.445678
2005-12-04.12.23.45.550000
2005-12-04.11.23.45.111111
%f: 12-04-2005.11 AM 23.45.111111
%f: 12-04-2005.11 AM 23.45.111111
%0f: 12-04-2005.11 AM 23.45.111111
15:21:38:000
15:21:38:000
1977-03-20.15.21.38.321678
1977-03-20.15.21.38.003216
1977-03-20.15.21.38.003216
03-20-1977.15.21.38.3216
```

Character String Functions

Use these functions to modify a character string (any valid character value). All character functions return a character value except for STRLEN, STRPOS, and VERIFY, which return an integer. In addition to the character string functions described in the syntax below, you can use the concatenation function to combine two character strings (see ++ (Concatenation) for more information).

Character String Function Syntax

character_string_function:

  character_string_function_with_one_parameter

  character_string_function_with_two_parameters

  substring_function

  conversion_of_numeric_data_to_char

character_string_function_with_one_parameter:

  char function_1_name '(' character_expression ')'

char_function_1_name:

  one of RTRIM UPPER LOWER STRLEN

character_string_function_with_two_parameters:

  char_function_2_name '(' character_expression, character_expression ')'

char_function_2_name:

  one of STRPOS VERIFY

substring_function:

  SUBSTR '(' character expression, num_expression [, num_expression] ')'

conversion_of_numeric_data_to_char:

  CHAR '(' num_expression [, format_string] ')'

format_string:

  character_expression

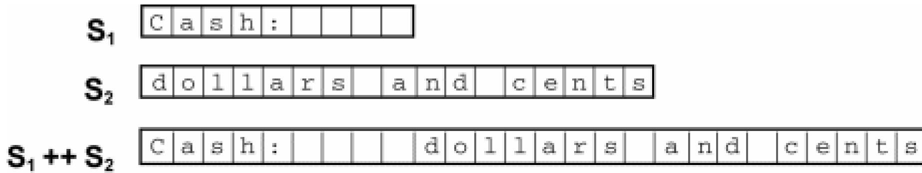
where:

- character_expression — see Character Expressions.
- num_expression — see Numeric Expressions.

++ (Concatenation)

This function returns the concatenation of the two input strings.

For example, concatenating a string having the value "Cash: " (with four trailing blanks) with a string having the value "dollars and cents" returns a string having the value "Cash: dollars and cents".



MIXED and DBCS values can be operands of concatenation in Java, ClassicCOBOL, and OpenCOBOL. The following table shows type and size of concatenation result for different platforms:

Mixed and DBCS Operands

Operands	Result type and size in Java	Result type and size in ClassicCOBOL and OpenCOBOL
CHAR(n) ++ CHAR(m)	CHAR(n+m)	CHAR(n+m)
DBCS(n) ++ DBCS(m)	DBCS(n+m)	DBCS(n+m)
DBCS(n) ++ CHAR(m)	MIXED(n+m)	MIXED(2*n+m+2)
DBCS(n) ++ MIXED(m)	MIXED(n+m)	MIXED(2*n+m+2)
MIXED(n) ++ MIXED(m)	MIXED(n+m)	MIXED(n+m)
MIXED(n) ++ CHAR(m)	MIXED(n+m)	MIXED(n+m)

where:

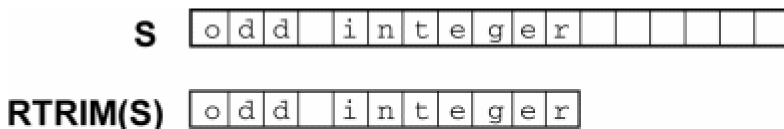
- n and m are length of character value.

For more information about the concatenation of unsigned integer pictures, see the PIC description.

RTRIM

This function returns the input string with any trailing blanks removed.

For example, trimming a string having a value "odd integer   " (five trailing blanks), returns a string having the value "odd integer" (no trailing blanks).



On some platforms, this function can be applied to MIXED and DBCS strings. See Restrictions on Features. For DBCS strings, double-byte trailing blanks are removed; for MIXED strings, both single-byte and double-byte trailing blanks are removed.

For specific considerations, refer to the following:

- RTRIM in Java
- RTRIM in ClassicCOBOL
- RTRIM in OpenCOBOL

UPPER and LOWER

These functions return the input string with all alphabetic characters converted to uppercase or lowercase respectively. The resulting string remains the same type, size, and length.

For example,

`UPPER ('12 E 49th Street')` returns `'12 E 49TH STREET'`

`LOWER ('12 E 49th Street')` returns `'12 e 49th street'`

On some platforms, these function can be applied to MIXED and DBCS strings. See Restrictions on Features.

Characters are converted to uppercase according to the specified codepage. For additional information, refer to Supported Codepages.

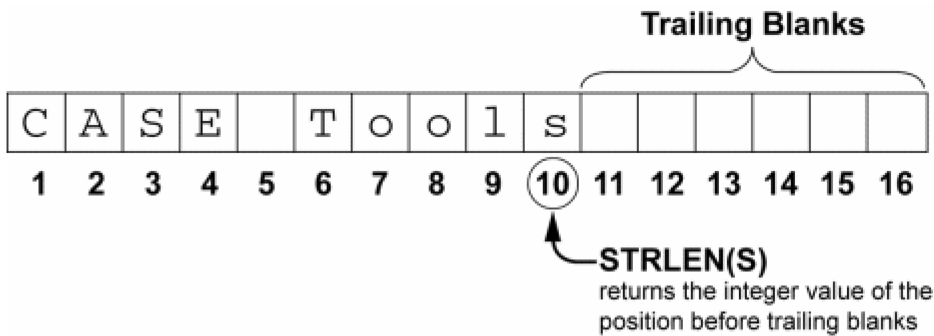
For specific considerations, refer to the following:

- UPPER and LOWER in Java
- UPPER and LOWER in ClassicCOBOL
- UPPER and LOWER in OpenCOBOL

STRLEN

This function returns a positive integer that specifies the length of the input string, not counting any trailing blanks. If the input string is all blanks or null, STRLEN returns a value of zero (0).

For example, using STRLEN on a string having a value "CASE Tools   " (with six trailing blanks) returns the integer value 10, which is the last non-blank position in the character string.



On some platforms this function can be applied to MIXED and DBCS strings. See Restrictions on Features. For DBCS and MIXED strings, it returns its length in characters not bytes; this is true for both single-byte and double-byte.

For specific considerations, refer to the following:

- STRLEN in Java
- STRLEN in ClassicCOBOL
- STRLEN in OpenCOBOL

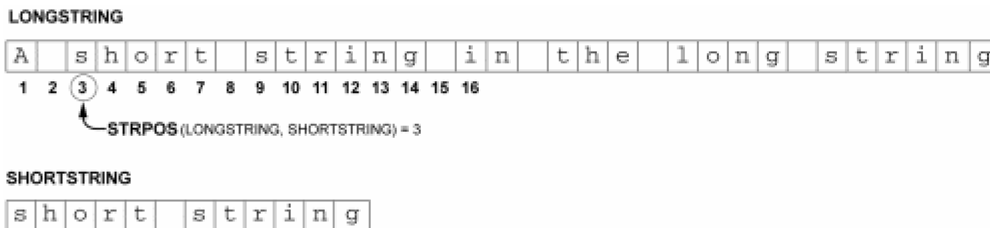
STRPOS

The STRPOS function searches for a second string in the first string and returns the position from which the second string starts. If the second string occurs more than once in the first string, the position of the first occurrence is returned. A zero is returned if the second string is not in the first string. This function is case-sensitive.

For example, if the value of LONGSTRING is "A short string in the long string" and the value of SHORTSTRING is "short string", then

```
STRPOS (LONGSTRING, SHORTSTRING)
```

returns the integer value 3, which is the position in LONGSTRING that contains the first character of SHORTSTRING.



On some platforms, this function can be applied to MIXED and DBCS strings. See Restrictions on Features. The position returned is the number of characters not bytes.

The following parameter types are accepted:

- STRPOS(char, char)
- STRPOS(dbcs, dbcs)

- STRPOS(mixed, char)
- STRPOS(mixed, mixed)
- STRPOS(mixed, dbcs)

A zero is returned if the second string is an empty string. In particular, STRPOS(s1,s2) = 0 if both s1 and s2 are empty strings.

For specific considerations, refer to the following:

- STRPOS in ClassicCOBOL
- STRPOS in OpenCOBOL

Position returned by this function measured in characters, not bytes.

VERIFY

This function looks for the first occurrence of a character in the first string that does not appear in the second string. Position returned by this function is measured in characters, not bytes. If all characters from the first string are found in the second string, then 0 is returned. The order of characters in the second string and the number of times one of those characters appears in the first string is irrelevant. This function is case-sensitive.

For example, if the variable NUMBERS_AND_SPACE is "0123456789 " (containing a space after the 9), then

```
VERIFY ('8000 Main Street', NUMBERS_AND_SPACE)
```

returns the position of the first character in the indicated string that is not a number or space. In this case, the integer value 6 is returned, which is the position of the M.

On some platforms, this function can be applied to MIXED and DBCS strings. See Restrictions on Features.

The following parameter types are accepted:

- VERIFY(char, char)
- VERIFY(dbcs, dbcs)
- VERIFY(mixed, char)
- VERIFY(mixed, mixed)
- VERIFY(mixed, dbcs)

For specific considerations, refer to the following:

- VERIFY in Java
- VERIFY in ClassicCOBOL
- VERIFY in OpenCOBOL

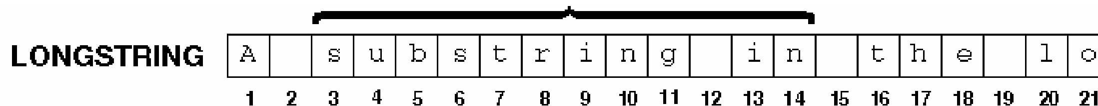
SUBSTR

The SUBSTR function returns a substring of the input string that begins at the position the first expression indicates for the length the second expression indicates. That is, the first expression is a positive integer that specifies the substring's starting position in the character string; that character becomes the first character of the resulting substring. The second expression is a positive integer that specifies the number of characters desired in the resulting substring. If the second expression is omitted, all the characters are copied from the specified starting position to the end of the string.

For example, if the value of LONGSTRING is "A substring in the long string", then

```
SUBSTR (LONGSTRING,3,12)
```

returns a string having the value "substring in", which includes the twelve characters starting at the third position in LONGSTRING.



On some platforms, this function can be applied to MIXED and DBCS strings. See Restrictions on Features. In this case position and length must specified in characters not bytes.

For specific considerations, refer to the following:

- SUBSTR in Java
- SUBSTR in ClassicCOBOL

- SUBSTR in OpenCOBOL

CHAR

The CHAR function supports conversion from numbers to character strings.

The following table outlines the characters that can appear in a format string, descriptions and examples of each.

CHAR Symbols and Descriptions

Symbol	Description	Examples	Value	Formatted Value
9	Echoes any digit (0-9)	9999.99 999.99	123.4 1234.543	0123.40 1234.54
$	Echoes "$" [3]	$9999	1234	$1234
Z or z	Echoes any digit (1-9) and removes leading "0". Trailing zeros that are within the format character range on the right side of the decimal separator is not suppressed.	ZZZ99.ZZ	123.4	123.40
-	Echoes a minus (-) for negative numbers. Nothing is printed for positive numbers	-9999 -9999 9999-	-1234 1234 -1234	-1234 1234 1234-
+	Prints positive numbers with a plus (+) sign, minus (-) sign is printed for negative numbers	+9999 +9999	1234 -1234	+1234 -1234
S or s	Country specific for C runtime. Java supplies country-specific values. They are used for Java applications.	S9999.99 C runtime: negative number format is set to (1.1) – number enclosed in parenthesis. Java runtime: Country Germany	-1234.12 1234.12 -1234.12 1234.12	(1234.12) 1234.12 -1234,12 1234,12
CR or cr	Negative numbers get "CR" suffix. Nothing is printed for non-negative numbers	9999CR 9999cr 9999cr 9999cr	-1234 -1234 1234 0	1234CR 1234cr 1234 0000
DB or db	Negative numbers get "DB" suffix. Nothing is printed for non-negative numbers	9999DB 9999db 9999db 9999db	-1234 -1234 1234 0	1234DB 1234db 1234 0000
*	"Check protection" character to avoid leading spaces	**99	123	*123
.	Decimal separator (country settings are used, they are read from the system for C and from Java settings for Java)	999.99	12.3	012.30
V or v	Decimal separator (country settings are used)	999V99	12.3	012.30
,	Thousands separator (country settings are used)	9,999v99	1234.5	1,234.50
Any other symbol	Echoes the symbol	(~S999.99) C runtime: negative number format is set to (1.1) – number enclosed in parenthesis.	-123.4	(~(123.40))

3. In Java, the currency symbol of current locale will be printed.

Most of the character function results do not have a predetermined length; instead, the AppBuilder environment determines the length at runtime.

The null string has a length of zero and is denoted with two single quotation marks without an intervening space (`''`).

Calling a character function does not change the value of the original character string.

For Java specific considerations, see CHAR in Java.

CHAR function with one argument

If no format string is provided to the CHAR function, it formats the number according to the following rules:

- CHAR applied to INTEGER or SMALLINT data items returns string number representation prefixed with '-' sign, if the argument is negative.
- CHAR applied to DEC and PIC data items returns string number representation containing no leading or trailing zeros prefixed with '-' sign, if the argument is negative. If the integer part of the number is zero, the result string starts with "0." or "-0." depending of the argument sign.
- If CHAR function argument contains invalid (overflowed) DEC or PIC data item, the result is empty string.
- CHAR applied to zero argument of any numeric type returns "0" string.

Example: CHAR function with one argument

`CHAR( -11.7107 )` returns `"-11.7107"`
`CHAR( 11.7107 )` returns `"11.7107"`
`CHAR(0.7107 )` returns `"0.7107"`
`CHAR( -0.7107 )` returns `"-0.7107"`
`CHAR( -1107 )` returns `"-1107"`
`CHAR( 1107 )` returns `"1107"`

Format string validation

Format string validation occurs for numeric functions only, meaning that it should be applied to a string only in case where the first parameter is numeric. The validation is performed when format strings are represented by string literals (not variables).

For the validation of a format string (when compiling) there are two possible choices, both involving the use of CHECK_DEC_FORMAT codegen parameter (for functions such as CHAR, INT and DEC):

- Preparation time validation
- Runtime validation.

CHECK_DEC_FORMAT controls only the validation for numeric functions.

- ***Preparation time validation*** examines the string for conformity with the new format string specification. The corresponding flag that controls this validation is CHECK_DEC_FORMAT from [CodegenParameters] section of hps.ini.

  When set to YES (default value), format string validation is performed.

- When ***runtime validation*** is performed, new runtime for CHAR function is used (i.e. new algorithm for dec to string conversion works). Runtime format string validation is controlled by CHECK_DEC_FORMAT flag from [AE Runtime] section of hps.ini that uses C runtime. Runtime validation for Java is controlled by CHECK_DEC_FORMAT parameter from [NC] section of appbuilder.ini.

  The default value is NO. When set to YES, the new algorithm for CHAR function is used.

CHAR Symbols and Descriptions when CHECK_DEC_FORMAT is set to YES illustrates the results returned by CHECK_DEC_FORMAT, when set to YES. For the results returned by CHECK_DEC_FORMAT when set to NO, see CHAR Symbols and Descriptions.

CHAR Symbols and Descriptions when CHECK_DEC_FORMAT is set to YES

Symbol	Description	Example	Result
9	Echoes a digit	char(123,"9999")	0123

Z z	Echoes a digit but removes leading zeros. These symbols can be used only before first occurrence of symbol 9 and only before decimal separator. **'' and 'z' 'Z' symbols are mutually exclusive. If ''** is used in a string, then no 'z' or 'Z' symbol is allowed, and vice versa. Note: While the AB2033 *Rules Language Manual* indicates that Zs after the decimal position are accepted, they are technically invalid and thus fail verification.	char(123,"zZ99") char(0,"z9z") char(0,"error:.zzz")	123 Error: "Wrong format string. Z (z) is not allowed after 9" Error: "Wrong format string. Z z * are not allowed after decimal separator"

*	"Check protection" symbol.
	
	Echoes a digit, but replaces leading zeroes with **'' characters. These symbols can be used only before the first occurrence of symbol 9 and only before decimal and thousands separators. ''** and 'z' 'Z' symbols are mutually exclusive.
	
	If '*' is used in a string, then no 'z' or 'Z' symbol is allowed, and vice versa.
	If '*' is used, then thousands separator is preserved in the output string.
		char(123,"**99")
		
		char(0,"z*9")
		
		char(0,"error:.zz***")
		
		`DCL`
		`   vch1 varchar (20);`
		`   d1 dec (12,2);`
		`ENDDCL`
		
		`map 12345.67 to d1`
		`map char(d1, "*.99")`
		`to vch1`
		`trace(vch1)`
		`map char(d1,`
		`"***,***,***,***.99")`
		`to vch1`
		`trace(vch1)`
			*123
			
			Error: "Z (z) and * are alternative."
			
			Error: "Wrong format string.
			
			Z z and * are not allowed after decimal separator"
			
			This rule output must be:
			12345.67
			**,
			**,*12,345.67

9 Z z *	Digit symbols No other format symbols except decimal separator and thousand separators can be located between first digit symbol occurrence and last one. Other symbols (non format) are allowed. If none of the digit symbols appears in the format string it implies that 'Z' symbol is located at the end of the format string and a warning is logged by Codegen.	char(123,"99DB99") char(8124284896, "phone:(999) 999-99-99") char(8124284896, "phone:(999) 999_99_99") char(1230771,"ID:zz9999_9999")	Error: Wrong format token "DB" in a digit sequence. Error: '-' is a format token. It is not allowed between signed digits. phone:(812) 428_48_96 ID: 0123_0771

V v .	Decimal separators Country specific decimal separator. Only one decimal separator is allowed per format string. A warning is generated in the case when there is no digit symbol occurrence before decimal separator. In this case it implies that the 'z' symbol is located just before decimal separator.	char(123.1,"Price:9,999v99.") char(123,"~~.999")	Error: Only one decimal separator is allowed. ~~123.000 Warning: If no digit symbols are found before the decimal separator, Z is assumed.
S s	Country specific sign.	char(-123,"s999")	-123 (United States)
+ -	Sign For "' format token, '" is printed for positive and "~~' for negative. In case of '~~ ' format token, nothing is printed for positive numbers and '-' is printed for negative.	Char(+123,"+999") char(-123,"999+") char(+123,"999-") char(-123,"-999")	+123 123- 123 -123
S s - +	Signs Only one sign is allowed per format string. Sign symbol can be located in any position of a format string before the first digit symbol occurrence or after the last one.	char(123,"State:-999") Character S is interpreted as symbol sign: char(-123,"s[999.999]")	Error: Only one sign is allowed per format string. -[123.000]
CR cr DB db	Credit/Debit symbols Any of these format tokens can be located in any position before the first digit sign occurrence or after the last one. Negative numbers can have these suffixes. Nothing is printed for non-negative value. Only one of these tokens can appear in the format string.	char(123,"9999cr") char(-123,"9999cr") char(123,"9999db") char(-123,"9999db") char(-123,"cr9999db")	0123 0123 cr 0123 0123db Error: Only one Credit/Debit symbol is allowed per format string.
, (comma)	Thousand separator Country specific thousand separator. This symbol should be surrounded by digit signs. It can be located only before decimal separator. The country specific thousand separator is generated only when a digit symbol exists to the left of this symbol.	Char(123456,"999,999,999") char(123456,"ZZZ,ZZZ,999") char(123456,"ZZZ,,ZZZ,,999") char(123456,"ZZZ,ZZZ.99,99")	000,123,456 123,456 Error: Wrong position of comma Error: Wrong position of comma
$	Country specific currency symbol Country specific currency symbol is printed. Only one currency symbol is allowed per format string. It can be located in any position before first digit symbol or after last digit symbol.	char(1,",***999$CR") char(1,"$zzz$")	***001$ Error: only one currency symbol is allowed.
Other Symbol	Symbols that are not described above. Echoes the symbol.	Char(8124284896, "phone:(999) 999_99_99")	phone:(812) 428_48_96

- If the number of digits in the integer part of formatted value exceeds the number of *digit symbols* used before decimal separator in the format string (i.e. some of the leading digits do not have associated any format symbol), then the first *digit symbol* is used to display these leading digits.

  For the locales where negative numbers are enclosed in the parenthesis, the position of the "s" or "S" symbol denotes the position of the left parenthesis. Right parenthesis is printed after the last *digit symbol* or after the *currency symbol*, if this one immediately follows the last *digit symbol*.

  Example: `CHAR (-13.45, "SZ99.99$")` will produce `"(13.45 )"` for some European countries (Germany, for example).

- Call of CHAR function with empty format string is equivalent to call of CHAR function with one argument.

  Example: `CHAR(123,"")` will issue the following result `123 Warning: Empty format string.`

List of Error Situations and Warnings

According to the format string specifications, the list of *error situations* is the following:

- Any format symbol, except 9 z Z * . V v . , between the first and the last digit symbol.
- More than one decimal separator.
- More than one sign symbol.
- More than one Credit/Debit symbol.
- More than one currency symbol.
- Wrong position of comma (used after decimal separator).
- 'Z' 'z' '*' occurrence after '9' token.
- 'Z' 'z' '*' occurrence after decimal separator.
- 'Z' ('z') and '*' symbols cannot be used together.

Warnings will be issued in the following situations:

- Format string is empty.
- No digit symbols found in the integer. Empty whole part.
- No digit signs found in a format string.
- Format string does not contain any of the format symbols.

Format String Validation Error Handling

Some format strings might cause preparation errors because they are considered invalid by the new rules introduced for format string verification. In order to make them preparable, you should set CHECK_DEC_FORMAT to NO.

However, for the successfully prepared and executed applications written prior to AppBuilder 2.1.3, using format strings considered invalid according to the new rules, you should set CHECK_DEC_FORMAT to YES. If you receive an error message, do one of the following:

1. Set Codegen and Runtime flag to NO. In this case, the application executes as it was prior to AppBuilder 2.1.3, fully backward compatible.
2. Try to fix the format string and continue to use preparation settings switched to YES. Please notice that in this particular case additional testing might be required, if the format string is a variable. This means it cannot be verified at preparation time and any possible error will be issued only at runtime.

Double-Byte Character Set Functions

The DBCS-enabled versions of AppBuilder also include three functions that cause AppBuilder to treat a character value of one data type as though it were a character value of another data type:

- CHAR (*character_value*)

  Treats the character value as a CHAR data item

- MIXED (*character_value*)

  Treats the character value as a MIXED data item

- DBCS (*character_value*)

  Treats the character value as a DBCS data item

You can have any character data types as arguments to these functions. For example, you can map the DBCS function applied to any string literal containing a valid DBCS value into a field of type DBCS.

If you provide these functions with a string literal, they are verified during the preparation process. If you provide them with a variable, they are not checked until execution. You will get an exception at runtime if a character string contains DBCS characters, which are not valid in runtime codepage.

You can also use the other character functions with a field of a DBCS or MIXED data type. See DBCS and MIXED Data Types for more information about the use of these data types.

Validation and Implementation of Double-Byte Character Set

Conversion functions MIXED and DBCS perform validations of their arguments. They determine whether or not the argument is actually a valid MIXED or DBCS value according to the specified codepage.

Refer to the topics below for more information:

- Double-Byte Character Set Functions in Java
- Double-Byte Character Set Functions in ClassicCOBOL and OpenCOBOL

Using CHAR, MIXED, and DBCS Data Types

When using variables and literals of CHAR, MIXED, and DBCS in the relational conditions, any combination of the operands in the relational condition is allowed; that is, each operand can be a variable or a literal of any character type: MIXED, CHAR or VARCHAR; however, a DBCS variable or literal can only be compared to another DBCS variable or literal. A warning is generated in the case of incompatible types of operands. See also Comparing Character Values in the Conditions section.

When using variables and literals of CHAR, MIXED, and DBCS in a MAP statement, the following rules apply:

- If the map destination is CHAR or VARCHAR variable, then the source can be CHAR, VARCHAR or MIXED variables or a literal of any character type. MIXED and DBCS literals are implicitly converted to CHAR data type.
- If the map destination is DBCS variable, then the source can be DBCS variable or DBCS literal.
- If the map destination is MIXED variable, then the source can be a variable or literal of any character type.

Error-Handling Functions

Three functions can be used to analyze errors during program execution:

- HPSError Function
- HPSResetError Function
- HPSErrorMessage Function

See Supported Functions by Release and Target Language for restrictions.

HPSError Function

The HPSError function returns an integer value. If the value is 0, then no error occurred. Otherwise, the value is the error code of the first error that occurs. The error code is set for arithmetic operations (division by zero, numeric overflow, and so on), for method invocations of AppBuilder-supplied window objects, for memory allocation problems, and for some other runtime errors.

To check for an error, do the following:

```
IF HPSError <> 0
    PERFORM ErrorHandler
ENDIF
```

If there are several errors, then the error code corresponds to the first error that occurred. Until the error code is reset with HPSResetError, this function returns the same value. Refer to the *Messages Reference Guide* for a list of the error codes returned by HPSError, along with the associated text strings returned by HPSErrorMessage.

HPSResetError Function

The HPSResetError function resets the error code to 0 after one or more error conditions have occurred.

HPSErrorMessage Function

The HPSErrorMessage function takes an error code as an argument and returns the text string containing a short description of the error condition. If an error description is not found, then the string is empty.

Refer to the *Messages Reference Guide* for a list of the error codes returned by HPSError, along with the associated text strings returned by HPSErrorMessage.

Support Functions

These miscellaneous functions support various features of AppBuilder.

Support Functions Syntax

support_function:

support_function_without_parameters

data_item_support_function

set_support_function

expression_support_function

field_support_function

view_support_function

support_function_without_parameters:

one of HIGH_VALUES LOW_VALUES SET_ROLLBACK_ONLY GET_ROLLBACK_ONLY GETRULESHORTNAME GETRULELONGNAME GETRULEIMPNAME

data_item_support_function:

data_item_supp_func_name1 ‘(‘ data_item ‘)’

data_item_supp_func_name1:

one of SIZEOF LOC

set_support_function:

SETDISPLAY ‘(‘ set_name, expression [, language] ‘)’

SETENCODING ‘(‘ set_name, character_exppression [, language] ‘)

expression_support_function:

TRACE ‘(‘ *expression* (, *expression*)* ‘)’

field_support_function:

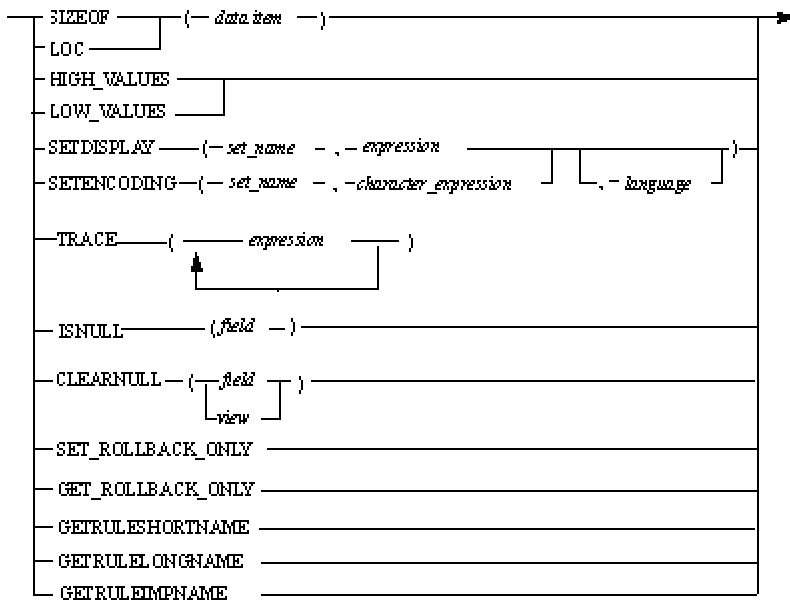
field_supp_func_name1 ‘(‘ *field* ‘)’

field_supp_func_name1:

one of ISNULL CLEARNULL

view_support_function:

CLEARNULL ‘(‘ *view* ‘)’



where:

- *character_expression* — see Character Expressions.
- *expression* — see Expression Syntax.
- *view* — see View.
- *data item* — see Data Items.

See Platform Support and Target Language Specifics for language-specific considerations on some of the functions.

SIZEOF

This function takes any Rules Language data item as its argument and returns its length in bytes in a field of type INTEGER.

SIZEOF Built-in Function Values by Platform

The following table lists the sizes of each Rules Language data type.

Platform-specific Data Type Sizes

Data Type	C	C#	Java	ClassicCOBOL	OpenCOBOL
BOOLEAN	2	2	2	2	2
SMALLINT	2	2	2	2	2
INTEGER	4	4	4	4	4
PIC (signed)[4]	Length+1	Length+1	Length+1	Length+1	Length+1
PIC (unsigned)[4]	Length	Length	Length	Length	Length
DEC (length, scale)	Length+1	Length+1	Length+1	Length div 2 + 1	Length div 2 + 1
DATE	4	4	4	4	10
TIME	4	4	4	4	12
TIMESTAMP	12	12	12	12	26
TEXT	256	256	256	256	256
IMAGE	256	256	256	256	256
OBJECT	-	4	4	-	-
CHAR (length)	Length	Length	Length	Length	Length

VARCHAR (length)	Length+2	Length+2	Length+2	Length+2	Length+2
DBCS (length)	Length*2	Length*2	Length*2	Length*2	Length*2
MIXED (length)	Length	Length	Length	Length	Length
VIEW	Sum of SIZEOF applied to each field and each sub-view field				

4. Length is the number of digits in PIC's storage picture. For example, the length of PIC 'S999V99' is 5.

In C and ClassicCOBOL generations only view can be used as argument of SIZEOF function.

SIZEOF function in C and CLASSICCOBOL is supported only for views.

Example: Using SIZEOF Function

```
MAP SIZEOF (VIEW_1) TO SIZE_LONG OF HPS_READ_FILE_LOCATE_MODE_I
```

Using SIZEOF flag helps you to calculate the SizeOf function call as

`sizeof (one_view_occurrence)*number_of_occurence.`

If this flag is not used, then SizeOf is calculated as

`sizeof (one_view_occurrence).`

When applied to a view with more than one occurrence, the SIZEOF function produces different results in C than on other platforms, like in the example below, where, for the view V:

```
DCL
   i integer;
   v view contains i;
   vv view contains v(10);
ENDDCL
```

the SIZEOF function returns 4 (size of the INTEGER field) in ClassicCobol, OpenCobol and in Java modes, while in C it returns 40, which is 4 multiplied by the number of occurrences, i.e. 10.

Without the SIZEOF flag, the SIZEOF of a view is NOT multiplied by the number of occurrences (in the previous example, the result will be 4 in C mode too); with this flag specified, the result is always multiplied by the number of occurrences and is 40 for all the platforms.

LOC

The LOC function takes a view as an argument and returns its location in a CHAR (8) field. It is used in conjunction with the following system components: HPS_READ_FILE_LOCATE_MODE, HPS_WRITE_FILE_LOCATE_MODE, and several HPS_BLOB components.

```
MAP LOC (VIEW_1) TO LOCATE_RECORD OF HPS_READ_FILE_LOCATE_MODE_I
```

In ClassicCOBOL, OpenCOBOL, and C, this function performs only object reference mapping, as shown in the example below:

```
dcl
   o object;
   i integer;
enddcl

map LOC(i) to o
```

LOC function accepts the parameter of any Rules Language type.

See the *System Components Reference Guide* for more information.

For Java specific considerations, see LOC in Java.

HIGH_VALUES

The HIGH_VALUES represents one or more characters that have the highest ordinal position in the collating sequence used and is useful for initializing database fields or for comparisons. Although designed for the mainframe LOCATE I/O mode system components, you can use it in all operating environments; however, you can map its value only to a field of type CHAR or VARCHAR.

Example: Using HIGH_VALUES Function

```
MAP HIGH_VALUES TO HI_CHAR_FIELD
```

On some platforms exact value could be used instead of HIGH_VALUES. For example, '0xff' literal could be assumed to have the highest ordinal position for character data type. However, this makes code to be unportable to other platforms. Thus it is highly recommended to use HIGH_VALUES instead of exact values literals.

For instance, the following code:

```
dcl
   x char(1);
enddcl
map HIGH_VALUES to x
if x <> "\xFF"
    trace("error")
endif
```

will be successful if prepared for OpenCOBOL but will fail for Java.

However, the following code:

```
if x <> HIGH_VLAUES
    trace("error")
endif
```

will work correctly on all platforms.

LOW_VALUES

The LOW_VALUES represents one or more characters that have the lowest ordinal position in the collating sequence used and is useful for initializing database fields or for comparisons. Although designed for the mainframe LOCATE I/O mode system components, you can use it in all operating environments; however, you can map its value only to a field of type CHAR or VARCHAR.

Example: Using LOW_VALUES Function

```
MAP LOW_VALUES TO LO_CHAR_FIELD
```

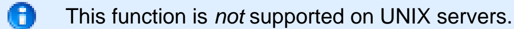
SETDISPLAY

The SETDISPLAY function supports the use of sets. Its first argument is the name of a Lookup Table set. Its second argument is the value to look up in the set and must be of the correct type for that set. The last argument is needed only for an Multiple Language Support (MLS) application and is the language entity to use for getting the representation of the encoding (the second argument). This argument defaults to the language entity of the active process.

This function returns a value in a CHAR (80) field even if the SET is MIXED or DBCS. You can use the corresponding conversion function to treat the returned value as a MIXED or DBCS value. If it does not find the encoding, it returns all spaces.

In Java, ClassicCOBOL, and OpenCOBOL, the lookup value can be MIXED or DBCS. Because this function looks for an identical value, MIXED with different types of trailing spaces or shift characters sequences are considered non-equal, and SETDISPLAY returns all spaces.

For additional considerations, see SETDISPLAY in ClassicCOBOL and OpenCOBOL.


This function is *not* supported on UNIX servers.

Example: Using SETDISPLAY Function

```
MAP SETDISPLAY (STATES_IN_US, 1) TO STATE_NAME
MAP SETDISPLAY (STATES_IN_US, OHIO IN STATES_IN_US) TO STATE_NAME
```

Example - Using SETDISPLAY Function with MIXED and DBCS Sets

```
MAP SETDISPLAY (MIXED_SET, 1) TO CHAR_VAR
MAP MIXED(CHAR_VAR) TO MIXED_VAR
MAP SETDISPLAY (DBCS_SET, 1) TO DBCS_VAR
MAP DBCS(RTRIM(CHAR_VAR)) TO DBCS_VAR
```

SETENCODING

The SETENCODING function supports the use of sets. Its first argument is the name of a Lookup Table set. Its second argument is the representation to look up in the set and can be any valid character value. In Java, ClassicCOBOL, and OpenCOBOL, the second argument can be MIXED or DBCS also. The last argument is needed only for an Multiple Language Support (MLS) application and is the language entity used for getting the display (the second argument). This argument defaults to the language entity of the active process.

SETENCODING returns a value of the same type and length as the set in the first argument. If that set is an INTEGER (31) set, the function returns an INTEGER (31) value. If it does not find the display, it returns zero (0) for a numeric set and all spaces for a character set. In Java, ClassicCOBOL, and OpenCOBOL, the representation value can be MIXED or DBCS. Since this function looks for an identical value, MIXED with different types of trailing spaces or shift characters sequences are considered non-equal, and SETENCODING will not find the corresponding encoding.

This function is *not* supported on UNIX servers.

Example: Using the SETENCODING Function

```
MAP SETENCODING (STATES_IN_US, 'OHIO') TO STATE_CODE
MAP SETENCODING (STATES_IN_US, STATE_NAME) TO STATE_CODE
```

TRACE

The TRACE function can be used to output the Rules Language data items to an application trace file. Fields, views, occurring views, and results of any expressions can be printed to the application trace. If the view is output to trace, field names, along with their values are output as well. The TRACE function has no return value and therefore, it cannot be used inside an expression.

The way each data type is printed with the TRACE statement depends on target language and might differ from one language to another. In other words, a variable in C generation can produce different output than same variable having the same value in OpenCOBOL generation.

See also TRACE in Java.

Example: Using TRACE Function

```
DCL
   I INTEGER;
   V VIEW CONTAINS I;
ENDDCL
MAP 27 TO I
TRACE(I)       *> Outputs "27"          <*
TRACE(I+3)     *> Outputs "30"          <*
TRACE(V)       *> Outputs "Field I: 27" <*
```

Example: Using TRACE Function with multiple arguments

```
DCL
   I INTEGER;
ENDDCL
MAP 27 TO I
TRACE(I, I+1)  *> Outputs in Java generation "27 28", in other generations "2728" <*
```

The output in this example depends on the generation: in Java generation, a space between two arguments is printed; in other generations, there is no space between arguments. That is in the example, the output in Java generation will be “27 28”, and the output in other generations will be “2728”.

GETRULESHORTNAME, GETRULELONGNAME, GETRULEIMPNAME

In order to capture the implementation name of the rule you are executing and to store it in error details, you can use the following functions:
`getRuleShortName()` – to get short name of the rule.
`getRuleLongName()` – to get long name of the rule.
`getRuleImpName()` – to get implementation name of the rule.

The corresponding macros are:
`CG_RULE_SHORT_NAME` – this macro is replaced by rule's long name.
`CG_RULE_LONG_NAME` – this macro is replaced by rule's long name.
`CG_RULE_IMP_NAME` – this macro is replaced by rule's long name.

For details about the use of these predefined macros, see also Using Name Macros.

Declarations

The name and data type of a variable or a procedure must be declared before it can be used. Use a declarative statement (DCL) to declare a:

- Local Variable Declaration
- Local Procedure Declaration
- Event Procedure Declaration

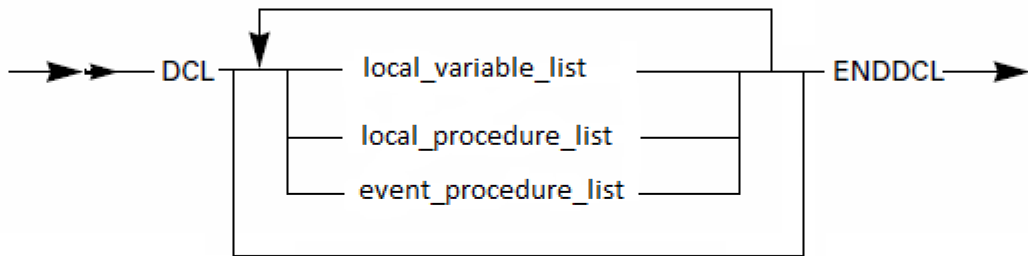
Declaration Syntax

declartion:

DCL (declarations_list)* ENDDCL

declarations_list:

one of local_variable_list local_procedure_list event_procedure_list



The following topics are also discussed in this chapter:

- Using Entities with Equal Names
- Choosing and Setting Signatures
- Using System Identifiers
- Controlling Compile Time Subscript

Local Variable Declaration

Use a DCL statement to declare a variable data item or view locally. A variable declared in a DCL statement is not contained in the repository, so it is not available to any rules or components the declaring rule uses. For usage information, review the following sections:

- Usage
- Valid Data Types
- Using LIKE Clause
- Using VIEW CONTAINS Clause

Local Variable Syntax

local_variable_list:

local_variable ; (local_variable ;)*

local_variable:

item_name (, *item_name*)* item_name_data_type

view_name (, *view_name*)* view_name_type

item_name_data_type:

data_type

LIKE variable_data_item

view_name_type:

LIKE *view_name*

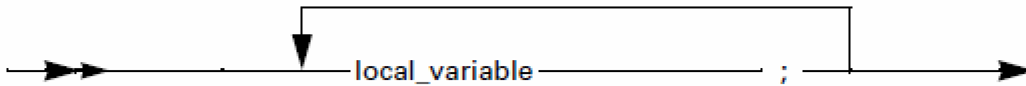
VIEW CONTAINS view_name_fields_and_views

view_name_fields_and_views:

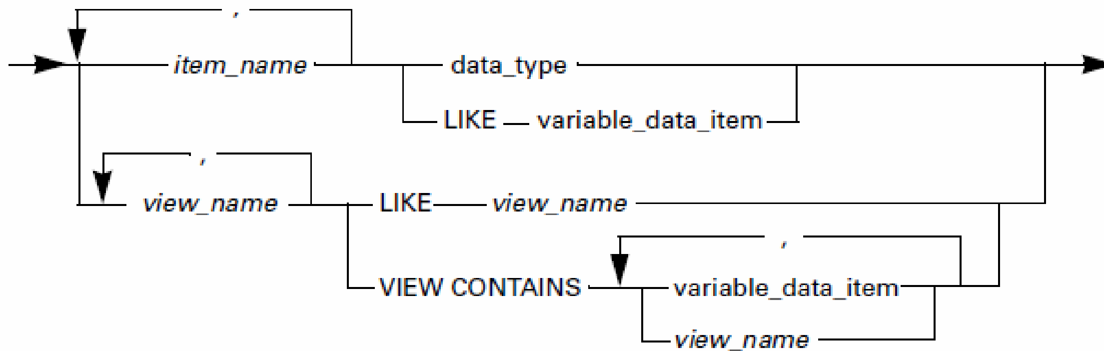
variable_data_item (, variable_data_item)*

view_name (, *view_name*)*

where local_variable_list is:



where local_variable is:



where:

- data_type — see Data Types.
- variable_data_item — see Variable Data Item.

An item_name or a view_name cannot start with the underscore symbol (_). Variable qualification (both OF and dot specification) cannot be used in LIKE clause.

Usage

Use a DCL statement at the beginning of a rule to declare variables local to the rule or inside a procedure to declare variables local to the procedure.

If you use a DCL statement at the beginning of a rule, the name of a locally-declared variable must not duplicate the name of any other variable in the data universe of the rule, either locally or in any subview. It also must not duplicate the class name used in the rule or the alias created with PRAGMA CLASSIMPORT.

If you use a DCL statement inside a procedure, the variable is local to the procedure even if the same name occurs elsewhere in the rule. That is, if a name is declared inside a procedure but also occurs outside the procedure, use of the name inside the procedure refers to the local variable and not to the variable existing outside the procedure.

You can have zero, one, or many DCL statements in a rule or procedure, but they must precede all other statements in that rule or procedure.

The DCL statement sets aside a temporary area of memory storage for use only during the execution of the declaring rule. When a rule or procedure is invoked, all locally declared fields are cleared (for example, character fields are set to the null string and numeric and date and time fields are set to 0). The reasons you might want to declare a data item locally include:

- Storing temporary data, such as during a swap procedure
- Breaking up data (for example, if you need only part of an employee record (say, the employee number) from a record in a flat file)
- Adjusting the size of a list box to reflect the number of records it contains

Valid Data Types

The following is a list of valid data_types that can be used in a variable declaration:

- character_data_type
- numeric_data_type
- data_and_time_data_type
- object_data_type
- boolean_data_type
- large_object_data_type

For a description of data_types, see Data Types

Using LIKE Clause

Use the LIKE keyword locally to define a field to be identical to another field in the data universe of the rule, or a view to be identical to another view in the data universe of the rule. Any variable after LIKE must have been declared previously, either locally or in a subview in the rule hierarchy, except the name of the new local variable.

If you declare a data item locally as being LIKE another, you can use the local data item exactly as you can use the original, but only within the declaring rule. A view declared locally has the same subviews and fields as the original view. A view declared with a LIKE clause can be subscripted so that it is multiple-occurring; a field cannot be subscripted.

Example: LIKE Keyword in DCL Statement

```
DCL
   COUNTER_1,COUNTER_2  SMALLINT;
   SUBTOTAL             INTEGER;
   NAME_TEMP            CHAR(30);
   ITEM_CODE            VARCHAR (20);
   PRICE                DEC (6,2);
   SHOW_PRICE           PIC '9999V99';
   DATE_OF_PURCHASE     DATE;
   TIME_OF_PURCHASE     TIME;
   CUSTOMER_TEMP (20)   LIKE NAME;  *> NAME is a view declared in rule hierarchy <*
ENDDCL
```

This DCL statement creates nine local fields and one local view. Each field can be used in the rule code exactly as if it were a field entity of that type in the data universe of the rule.

CUSTOMER_TEMP is declared LIKE NAME and is also declared as occurring 20 times. You can use it in the rule code exactly as if it were a multiple-occurring view with the same subviews and fields as the NAME view. You do not have to qualify the view name to specify one instance of NAME because you are referring to the single definition of the view in the repository.

If NAME is declared as a view or subview, then the code is syntactically correct. If NAME is a field, the preparation of the rule results in a syntax error because CUSTOMER_TEMP, as shown above, is subscripted, and a field cannot be subscripted.

If NAME is redefined by an R_NAME view and some subview of NAME called SUB_NAME is redefined by R_SUB_NAME view, then subview of CUSTOMER_TEMP called SUB_NAME is considered to be redefined by R_SUB_NAME view; however CUSTOMER_TEMP is not considered to be redefined.

Using VIEW CONTAINS Clause

Use a VIEW CONTAINS clause to build a local view from lower-level views and fields. Building a local view with this clause is equivalent to building a View *includes* Field or View *includes* View relationship in a repository. Unlike the name of the new local variable, any variable after VIEW CONTAINS must have been declared previously, either locally or in a subview. Thus, local declaration of views is "bottom up" or "inside out."

All the identifiers to the left of VIEW CONTAINS represent views, each of which relates to each of the field or subview entities to the right of VIEW CONTAINS. Order is important when building a view locally, just as it is when building a view within the repository. The order in which the elements appear indicates their position in the hierarchy of the view, with left most element being the highest in the hierarchy, down to the right most, which is lowest. However, the order in which those fields and subviews were originally built does not matter as long as they exist before you attempt to relate them to the higher-level view.

A VIEW CONTAINS clause has a parent view on the left and a child view or field on the right.

As discussed in Variable Data Item, you can omit the names of some of the ancestral views of a field in a statement. However, you must always include the occurrence number of a multiple-occurring subview in a statement even if the view name does not appear.

Local Procedure Declaration

Declaration of a local procedure in a DCL statement is useful if the procedure might be used before it is defined. For example, consider two procedures that call each other: the first one calls the second, which is not defined, its definition located below the first procedure. This situation can be resolved by declaring the second procedure in a DCL statement of the rule.

The declared procedure must be defined somewhere in the rule (where the procedure definition is allowed). See Common Procedure for the syntax diagram and description of a procedure.

Local Procedure Syntax

local_procedure_list:

local_procedure ; (local_procedure ;)*

local_procedure:

proc_name PROC [parameter_list] [: output_type]

parameter_list:

'(' *parameter_name* [parameter_name_type] (, *parameter_name* [parameter_name_type])* ')'

parameter_name_type:

data_type

LIKE *field_name*

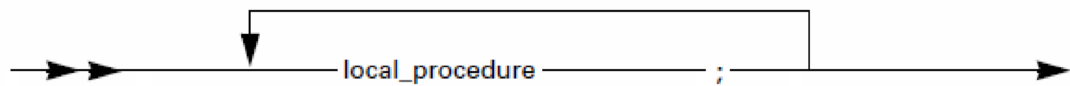
output_type:

data_type

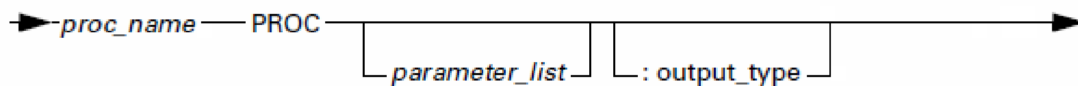
LIKE variable_data_item

LIKE *view_name*

where local_procedure_list is:

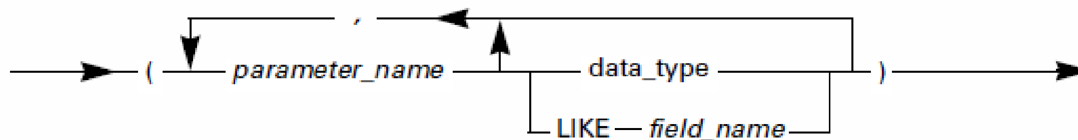


where local_procedure is:



where:

- *proc_name* is the name of a procedure to be declared.
- *parameter_list* is:



where:

- data_type — see Data Types.

- A proc_name or a parameter_name cannot start with the underscore symbol (_).
- The output type can be any data type, except objects.
- Alias declaration can never be used as a procedure formal parameter.
- For platform specific information, see Local Procedure Declaration in Java.

Example: Declared Procedure

```
DCL
   *> Procedures declaration <*
   PROC1 PROC(I INTEGER);
   PROC2 PROC(I INTEGER) : INTEGER;
ENDDCL

PROC1(10)

*> Procedures definition <*
PROC PROC1(I INTEGER)
   DCL
      Result INTEGER;
   ENDDCL

   MAP PROC2(I) TO Result
   PRINT Result *> "100" is printed <*
ENDPROC

PROC PROC2(I INTEGER) : INTEGER
   PROC RETURN(I*I)
ENDPROC
```

Event Procedure Declaration

An event procedure is invoked when an event you have chosen to respond to is triggered for an object. For a list of available events, see the *ObjectSpeak Reference Guide.*

Event Procedure Syntax

event_procedure_list:

event_procedure ; (event_procedure ;)*

event_procedure:

proc_name PROC FOR *event_name* [LISTENER *listener_name*] event_name_type

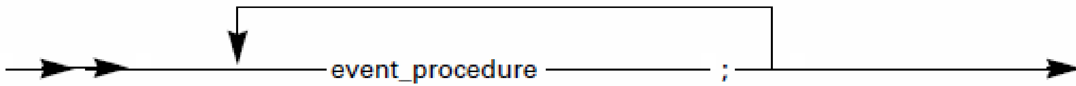
event_name_type:

OBJECT *object_name*

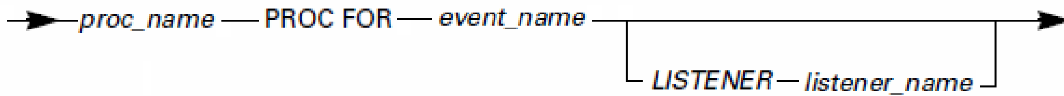
TYPE ' *class_identifier* '

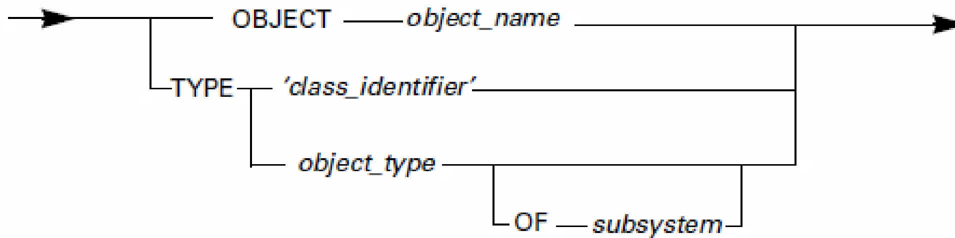
TYPE *object_type* [OF *subsystem*]

where event_procedure_list is:



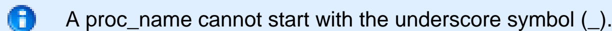
where event_procedure is:





where:

- *proc_name* is the name of a procedure to be declared.
- *event_name* is the name of the declared object event.
- *listener_name* is the name of the interface that implements event triggering (Java only).
- *object_name* can of any of the following:
    - The system identifier (HPSID) of the object.
    - The alias of the object — see Alias.
    - A pointer to the object — see Object Data Types.
- *class_identifier* is a string that identified the class. It might be CLSID or OLE objects or fully qualified class name for Java classes. The identification string is considered case-sensitive.
- *object_type* is the type of the object whose events the procedure receives — see Object Types.
- *subsystem* is the group that the object belongs to. The following are supported:
    - GUI_KERNEL, the set of window controls supplied with AppBuilder.
    - JAVABEANS, for any Java class.


A proc_name cannot start with the underscore symbol (_).

Object Types

An object_name cannot be the same as an AppBuilder predefined object_type name. A full list of object_type names can be found in the *ObjectSpeak Reference Guide.*

Usage

You must define an event procedure inside the rule that converses the window that contains the objects whose events the procedure responds to. See Event Handling Procedure.

For more details about the automatic handler assignment, see the Event Handler Statement in Java description.

A declaration (DCL) statement is *not* necessary to define an event procedure. Use a DCL statement to specify the same procedure for multiple events, objects, or object types. For example, suppose the following procedure is defined in a rule:

```
PROC clickControl
   ...
ENDPROC
```

Using this procedure and the appropriate DCL statement, you can use the procedure for:

- One Event Procedure for Multiple Events (for multiple events of the same object)
- One Event Procedure for Multiple Objects
- One Event Procedure for Multiple Object Types
- LISTENER Clause (Java only)

One Event Procedure for Multiple Events

If you want the same procedure to handle the Click and DoubleClick events for the same object, include the following DCL statement at the beginning of the rule that contains the procedure:

```
DCL
    clickControl PROC FOR Click OBJECT myListBox;
    clickControl PROC FOR DoubleClick OBJECT myListBox;
ENDDCL
```

One Event Procedure for Multiple Objects

If you want the same procedure to handle an event for multiple objects, include the following DCL statement at the beginning of the rule:

```
DCL
    clickControl PROC FOR Click OBJECT myListBox;
    clickControl PROC FOR Click OBJECT myPushButton;
    clickControl PROC FOR Click OBJECT myRadioButton;
ENDDCL
```

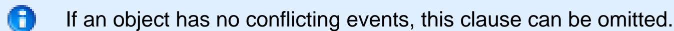
One Event Procedure for Multiple Object Types

If you want the same procedure to handle an event for multiple object types, include the following DCL statement:

```
DCL
    clickControl PROC FOR Click TYPE ListBox;
    clickControl PROC FOR Click TYPE PushButton;
    clickControl PROC FOR Click TYPE RadioButton;
ENDDCL
```

LISTENER Clause

The LISTENER clause is only available for Java. Use the LISTENER clause to avoid conflicting situations when an object has two events with the same name.



If an object has no conflicting events, this clause can be omitted.

See also Example: Java LISTENER Clause.

Using Entities with Equal Names

It is possible to have several different entities with the same name in a rule scope; however, in some cases, one entity might *hide* another (making that entity unusable in a rule), and in other cases, even when names of entities are the same, it is clear from the syntax which of the two must be used. For example, if there is a Procedure and a Field with the name "I", and there is a procedure call, it is clear that a Procedure must be used instead of a Field, as shown in the Example: Entities Using the Same Name.

In general, if two entities have the same order of precedence, their names must not coincide. For example, if a Rule object and a Window object, having the same order of precedence, have the same name, then the last one on the list becomes visible and the other becomes hidden.

If two entities have the same name and their order of precedence is not equal, the one that is higher in the order of precedence (that is, with smaller order number - see Entity Precedence) is selected and the other becomes inaccessible in the Rule; however, there are a number of exceptions to this rule as listed below:

- A Field can have the same name as a View if the Field is declared in the Rule hierarchy (not in the Rule code) and it is not a direct child of the View.

- A View and a Field can have the same name as a Set or Set Symbol, but only if it is declared in the Rule hierarchy. In this case, the Set object created for the Set cannot be used in the Rule.
- A Field and a View can have the same name as a Rule, but only if it is declared in the Rule hierarchy. In this case, the Rule object created for the Rule cannot be used in the Rule.
- A Set and Set Symbols can have the same name as a Rule. In this case, the Set object created for the Set cannot be used in the Rule.
- A Set and Set Symbol can have the same name, but in this case the Set object created for the Set cannot be used in the Rule.
- When the Window object, Rule object, System identifier (HPSID), MENUITEM or Set Object have the same name, then only the Window object can be used in the Rule.

AppBuilder follows the order of precedence in the following table.

Entity Precedence

Order	Entities
1	Fields, views, sets and set symbols
2	Procedures
3	Rules
4	Window object, Rule object, System identifier (HPSID), MENUITEM, Set Object

Example: Entities Using the Same Name

In this example, both the procedure and the field have the name "I".

```
DCL
   I, J INTEGER;
ENDDCL
PROC I : INTEGER
   *> ...some code... <*
ENDDPROC
MAP 1 TO I *> Variable <*
I  *> Procedure (return value is lost) <*
MAP I TO J *> Variable (according to order of precedence) <*
PERFORM I  *> Procedure (return value is lost) <*
```

Choosing and Setting Signatures

When choosing between a field and a procedure or between different procedures, the names and the signatures are compared to each other.

- The signature of a procedure is its name and a list of parameters.
- The signature of a field or view is its name, list of occurrence indices, and the qualification (of the view name).
- The signature of a set symbol is its name and the qualification (in the set name).

Two signatures are equal if the names and the number of parameters are equal, the parameter types are compatible, and the qualifications are the same.

The following table lists compatible types. Types listed on the same row are compatible.

List of Compatible Parameter Types

Parameter Types
INTEGER, SMALLINT
DEC, PIC
CHAR, VARCHAR, TEXT, IMAGE
DBCS
MIXED
DATE
TIME

TIMESTAMP
OBJECT

To distinguish between fields and set symbols with the same name, use a qualification. To distinguish between procedure, set symbol, field, or view, use a unique parameter type for each.

Although you cannot declare a local variable with the same name as a set symbol, you can use a field with the same name as a set symbol that exists in an external view created in a Hierarchy diagram.

Examples: Choosing and Setting Signatures

The following is an example of choosing and setting signatures:

```
DCL
   I INTEGER;
   V VIEW CONTAINS I;
   V1 VIEW CONTAINS V(10);
ENDDCL

PROC I(D DATE)        *> This procedure is visible - it has a different
                      *  signature with variable I <*
   ...
ENDPROC

PROC V1(I INTEGER)    *> This procedure is visible - as view V1
                      *  does not have subscripts it has a different
                      *  signature with this procedure <*
   ...
ENDPROC

PROC V1 : INTEGER     *> This procedure conflicts with view V1 -
                      *  see examples below on how to use it <*
   ...
ENDPROC

PROC V1(D DEC(10,1)) : INTEGER
   ...
ENDPROC

PROC V1(D DEC(10,2))  *> This causes a compile error:
                      *  V1(DEC(10,1)) may not be redefined
                      *  for example, in procedure call V1(1.1)
                      *  it is impossible to distinguish
                      *  which one is meant <*
   ...
ENDPROC

PROC V1(SI SMALLINT)  *> This causes a compile error:
                      *  V1(INTEGER) may not be redefined
                      *  for example, in procedure call V1(27)
                      *  it is impossible to distinguish
                      * which one is meant <*
   ...
ENDPROC

PERFORM V1                  *> Procedure V1 without parameters is called -
                             *  variable can not be used in PERFORM
                             *  clause <*
V1                           *> Procedure V1 without parameters is called - variable can not occur
here <*
V1(0)                        *> V1(INTEGER) is called, because argument is integer <*
V1(0.0)                      *> V1(DEC) is called because argument is decimal <*
MAP V1 TO SomeVariable       *> View V1 is mapped to SomeVariable,
                             *  according to order of precedence,
                             *  not return value of procedure V1 <*
MAP V1(1.1) TO SomeVariable  *> Here the return value of V1(DEC)
                             *  is mapped to SomeVariable <*
```

Example - Ambiguity Error

In the following example, it is impossible to use the return value of procedure V1.

```
*> Hierarchy:
   Set K
   Value J
   Value L
   Value M
   View IO_VIEW
   Field M
<*
DCL
   J INTEGER; *> Incorrect - same name as set symbol <*
   I, K INTEGER;
   V1 VIEW CONTAINS I;
   V2 VIEW CONTAINS I, K;
   L INTEGER; *> Incorrect - can't distinguish between set symbol and local variable <*
   DISPLAY CHAR(10);
ENDDCL

MAP 1 TO I          *> Ambiguity error - I OF V1 or I OF V2? <*
MAP 27 TO I OF V1   *> OK <*
MAP 15 TO M         *> OK - M is a variable, set symbol
                    *  cannot be used here <*
MAP 15 TO M OF IO_VIEW       *> OK <*
MAP M TO I OF V2             *> Ambiguity error - set symbol M or J OF V1? <*
MAP M IN K TO I OF V2        *> OK - M is a set symbol <*
MAP M OF IO_VIEW TO I OF V2  *> OK - I is a field <*
MAP K TO I OF V1             *> OK <*
MAP I OF V1 TO K             *> OK <*
MAP ROUND(K, -2) TO I OF V1  *> OK - set K can not be used in ROUND, so this
                             * line is equal to the next line <*
MAP ROUND(K OF V2, -2) TO I OF V2     *> OK <*
MAP SETDISPLAY(K, J IN K) TO DISPLAY  *> OK <*
```

Using System Identifiers

The system identifier (HPSID) that coincides with a name of some other entity is always hidden. However, the object with this HPSID can still be used if an alias is declared for this HPSID using an OBJECT 'HPSID' DCL statement clause.

The system identifier (HPSID) is case-sensitive, but other rules identifiers are not. If a system identifier differs from some other identifier only in case, then two identifiers are considered to be the same.

If there are two system identifiers (HPSIDs) with the same name or their names differ only in case, only one of them can be used in a rule. The one that can be used is chosen according to the order of precedence in the following list in descending order: (the system identifier in uppercase is chosen).

- WINDOW of GUI_KERNEL
- RULE of GUI_KERNEL
- Other object type
- MENUITEM of GUI_KERNEL

Controlling Compile Time Subscript

A subscript is a symbol or number used to identify an element in an array. At compile time, subscript control is performed for all constant subscripts in the rule. If a constant subscript is less than one, or it is greater than the view size, a preparation error occurs. (Refer to the *Messages Reference Guide* for descriptions of error messages). If the subscript expression contains a variable data item, it cannot be verified at compile time.

Refer to the following topics for descriptions of subscript control for different languages:

- Subscript Control in C
- Subscript Control in Java
- Subscript Control in ClassicCOBOL
- Subscript Control in OpenCOBOL

Subscript control at compile time does *not* perform in Java because of dynamic views support. See Dynamically-Set View Functions in Java for more information.

Example: Subscript Control

```
DCL
   I INTEGER;
   V(10) VIEW CONTAINS I;
   INDX INTEGER;
ENDDCL

MAP 1 TO I(0)      *> Compile time error, I(0) doesn't exist, subscript is less than one <*
MAP 0 TO INDX
MAP 1 TO I(INDX)   *> Runtime error <*
```

Preparing a Rule Declaration Example

In the following example of a local Rule declaration, VIEW_6 is a view that consists, among other items, of VIEW_2, which occurs 10 times within VIEW_6, of VIEW_1, and of VARCH_1. The rule also contains VIEW_2, which consists of PIC_1 and DEC_1 defined as PIC 'S999V99' and DEC (9, 3) fields. VIEW_1 consists of CHAR_1 and CHAR_2, which themselves are declared as CHAR (1) fields. VARCH_1 is a VARCHAR field of (maximal) length 6.

Views VIEW_3, VIEW_4, and VIEW_5 are all constructed the same way; each consists of FLD1 followed by FLD2. The rule does not prepare correctly unless FLD1 and FLD2 already exist in the data universe of the rule.

Sample Local Rule Declaration

```
DCL
   CHAR_1, CHAR_2 CHAR;
   VARCH_1 VARCHAR (6);
   PIC_1 PIC 'S999V99';
   DEC_1 DEC (9, 3);

   VIEW_1 VIEW CONTAINS
          CHAR_1,
          CHAR_2;
   *> Note the "Inside out" principle: The building blocks CHAR_1 and
   *  CHAR_2 must have been declared before the containing view VIEW_1
   *  can be declared <*

   VIEW_2 VIEW CONTAINS
          PIC_1,
          DEC_1;

   VIEW_3, VIEW_4, VIEW_5 VIEW CONTAINS
                          Fld1,
                          Fld2;

   VIEW_6 VIEW CONTAINS
          - - - -
          - - - -
          VIEW_2 (10),
          - - - -
          - - - -
          VIEW_1,
          VARCH_1;
ENDDCL
```

In this example, the following statement in the rule:

```
MAP VARCH_1 TO CHAR_2
```

is interpreted as:

```
MAP VARCH_1 OF VIEW_6
TO CHAR_2 OF VIEW_1 OF VIEW_6
```

In other words, there are no local variables CHAR_2, VIEW_1, or VARCH_1 that exist by themselves. They exist only as subviews of VIEW_6.

In addition, the following declaration:

```
DCL
   VIEW_7 VIEW CONTAINS
   CHAR_1, CHAR_2;
ENDDCL
```

is equivalent to:

```
DCL
   VIEW_7 LIKE VIEW_1;
ENDDCL
```

assuming that this declaration is made after that of VIEW_1.

The first MAP statement is now invalid because it is not clear whether CHAR_2 refers to CHAR_2 OF VIEW_1 OF VIEW_6 or to CHAR_2 OF VIEW_7. To avoid this kind of ambiguity, qualify a name. For example, CHAR_2 OF VIEW_6 is acceptable.

Setting Number of Occurrences

Right-Side Subscript

A subscript on the child view indicates the number of times that view occurs within the parent view. The child in this case can only be a View; it *cannot* be a Field.

In the following example, View A consists of 10 occurrences of View B:

```
DCL
   A VIEW CONTAINS B (10);
ENDDCL
```

Left-Side Subscript

A subscript on the child view or field indicates the number of times that view or field occurs within the parent view (*only* if this left side parent view does not itself become a building block for a higher-level view). If the parent view does become a part of a higher-level view, you are not prompted with a syntax error but subscripts are ignored.

Example: Assume that the following constitutes *all* local declarations for a rule.

```
DCL
   A (7) VIEW CONTAINS B;
   P (5) VIEW CONTAINS B;
   X (3) VIEW CONTAINS P;
ENDDCL
```

A is not used as a building block and you can reference A(1), A(2),..., A(7). P is used as a building block for X, so you cannot subscript P five times. X is not used as a building block and you can reference X(1), X(2), X(3).

However, if you assume that the last declaration is

```
X (3) VIEW CONTAINS P (4);
```

the left side subscripting P(5) is again ignored, but the right side subscripting P(4) is not. In this situation, you can qualify

```
P of X (m, n)
```

where 1 m 3 and 1 n 4.

Procedures

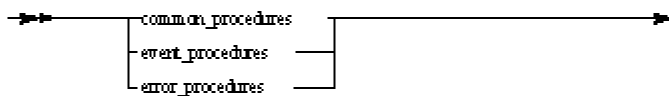
A procedure is defined using the PROC and ENDPROC keywords.

- Define a Common Procedure to encapsulate portions of code.
- Define a Event Handling Procedure to respond to events from objects.

Procedure Syntax

procedure:

one of common_procedure_def event_procedure_def error_procedure_def



Common Procedure

You can define a procedure anywhere within a rule except within another procedure body. However, because a procedure must be defined before you can invoke it, a natural placement for a procedure definition is near the top of a rule. See PERFORM Statement for information about invoking a procedure.

A procedure can consist of any number of Rules Language statements. You can invoke a procedure only in the rule in which the procedure is defined.

Common Procedure Syntax

common_procedure_def:

PROC common_procedure proc_statements ENDPROC

common_procedure:

procedure_name [parameter_list] [: output_type]

parameter_list:

'(' *parameter_name* (, *parameter_name*)* [parameter_name_type] (, *parameter_name* (, *parameter_name*)* [parameter_name_type])* ')'

'(' *view_name* (, *view_name*)* [LIKE *view_name*] ')'

parameter_name_type:

data_type

LIKE *field_name*

output_type:

data_type

LIKE variable_data_item

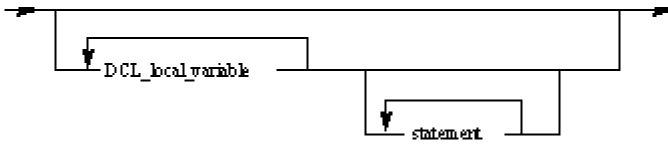
LIKE *view_name*

proc_statements:

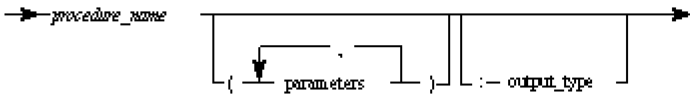
DCL local_variable_list ENDDCL statement_list

PROC common_procedure proc_statements ENDPROC

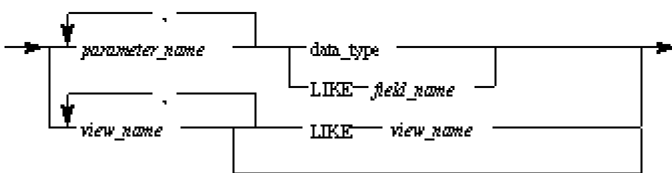
where proc_statements are:



where common_procedure is:

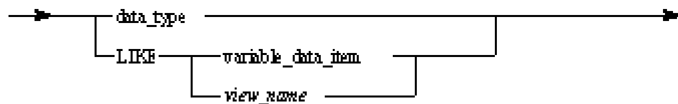


where *parameters* can be:



If the parameter view does not use the LIKE *view_name* clause, the view must be declared in the procedure's DCL section.

where *output_type* can be:



where:

- data_type — see Data Types.
- DCL_local_variable — see Local Variable Declaration.
- variable_data_item — see Variable Data Item. Note that OBJECT array cannot be a parameter.
- statement — any Rules Language statement, except procedure declaration.

The output type can be any data type, *except* objects.

Alias declaration can never be used as a procedure formal parameter.

Platform Specific Considerations

You cannot declare a procedure return result using the LIKE clause for ClassicCOBOL.

For additional information about common procedure parameters, see Defining Views in Java and Common Procedures in C.

Common Procedure Usage

You can pass individual data items, or literals as parameters to procedures. If you pass a view to a procedure, the view must be declared inside the procedure receiving it (see Example 3: Passing a View to a Procedure).

A procedure can return a value or a view. If the procedure returns a value, the procedure is treated like a function and can be used in any context in which a function can be used (see Example 2: Using a Procedure as a Function). At the preparation time AppBuilder verifies that all possible

execution paths return a value, if a procedure was declared as returning value. If the procedure returns a view, the view must also be declared in the rule DCL section or exist within the rule data hierarchy. If one of the procedure's parameters is a view that was not declared through the LIKE clause, it must be declared in the procedure. The four examples shown below illustrate how to use common procedures.

Examples: Common Procedures

Example 1: Simplifying Error Code Processing

By defining a procedure with code common to multiple error code processing, the coding of each process is simplified.

```
PROC handleError(errorCode SMALLINT)
DCL
   errorDescr VARCHAR(255);
ENDDCL
IF errorCode <= 0
   MAP "SUCCESS" TO errorDescr
   ELSE IF errorCode <= 2
      MAP "WARNING" TO errorDescr
   ELSE
      MAP "SEVERE ERROR" TO errorDescr
   ENDIF
ENDIF
PRINT errorDescr
ENDPROC
.
handleError(code)
.
handleError(dbCode)
```

Example 2: Using a Procedure as a Function

The procedure "cubed" receives one parameter (an integer) and returns the cube of that number. The procedure can be used in any context in which a function can be used — in this case in a MAP statement.

```
PROC cubed (inputNumber INTEGER): INTEGER
   PROC RETURN (inputNumber * inputNumber * inputNumber)
ENDPROC
.
MAP cubed(anyNumber) to y
```

Example 3: Passing a View to a Procedure

The procedure returns a numeric value:

```
PROC getTaxableIncome (income VIEW): DEC(31,12)
   DCL
      baseSalary DEC(31,2);
      bonus, commissions DEC(31,2);
      income VIEW CONTAINS baseSalary,bonus,commissions;
   ENDDCL
   PROC RETURN (baseSalary + bonus + commissions)
ENDPROC
.
MAP getTaxableIncome(income) * taxRate to tax
```

Example 4: Returning a View from a Procedure

```
DCL
   CUSTOMER_NAME CHAR(30);
   ORDER_NO INTEGER;
   ORDER_RECORD VIEW CONTAINS CUSTOMER_NAME, ORDER_NO;
   LAST_NO INTEGER;
ENDDCL

PROC CREATE_ORDER(NAME CHAR(30)) : LIKE ORDER_RECORD
   DCL
      V LIKE ORDER_RECORD;
   ENDDCL
   MAP LAST_NO + 1 TO ORDER_NO OF V, LAST_NO
   MAP NAME TO CUSTOMER_NAME OF V
   PROC RETURN (V)
ENDPROC

PROC NEW_ORDER(V LIKE ORDER_RECORD) : LIKE ORDER_RECORD
   MAP LAST_NO + 1 TO LAST_NO
   PROC RETURN ({ CUSTOMER_NAME OF V, LAST_NO })
ENDPROC
```

Event Handling Procedure

There are two ways to handle events:

- Event Procedures
- HPS_EVENT_VIEW Method

See also the following related information:

- Event Parameters
- Specific Restrictions for Constructing an Event Handler in C
- Specific considerations and restrictions for Constructing an Event Handler in Java

Event Procedure Syntax

event_procedure_def:

PROC event_procedure proc_statements ENDPROC

event_procedure:

proc_name [FOR *event_name* event_name_type] parameter_list

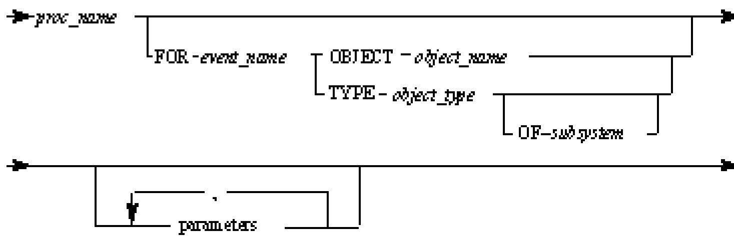
event_name_type:

OBJECT *object_name*

TYPE *object_type* [OF *subsystem*]

PROC — event_procedure — proc_statements — ENDPROC

where event_procedure is:



where:

- *object_name* can be any of the following:
  - The HPSID of the object.
  - The alias of the object — see Alias.
  - A pointer to the object — see Alias.
- data_type — see Data Types.
- variable_data_item — see Variable Data Item.
- parameters — see Common Procedure.
- proc_statements — see Common Procedure.
- *object_type* is the type of object whose event(s) the procedure receives.
- *subsystem* is the group to which the object pointed to belongs.

For a list of available object types, see the *ObjectSpeak Reference Guide*.

The following subsystems are supported:

- GUI_KERNEL is the set of window controls supplied by AppBuilder.
- JAVABEANS is used for any Java class.

It is necessary to specify a subsystem only if there is an ambiguity.

Event Procedures

To use an event procedure, include it in the rule that converses the window. Do *not* use Window Painter to select the events to be handled. You must write a procedure for an event to be handled.

An event procedure is invoked when an event is triggered for an object. When the event procedure finishes or when a PROC RETURN statement is encountered, the rule continues conversing the window, waiting for another user event to occur.

Control does not return to the statement following the CONVERSE until an event is returned in HPS_EVENT_VIEW. This includes any event triggered by AppBuilder-supplied (GUI_KERNEL) window control.

Event procedures are supported only for AppBuilder-supplied window objects in Java. To handle events from AppBuilder-supplied window objects in C, test the contents of HPS_EVENT_VIEW in statements following the CONVERSE statement.

You must define an event procedure inside the rule that converses the window containing the Java objects whose events the procedure responds to. All Rules Language statements are allowed inside event processes, *except*:

- CONVERSE
- USE RULE
- USE COMPONENT
- POST EVENT
- PERFORM

You cannot modify a window view (a view whose parent is a window) inside an event procedure for that window. To perform tasks that are not allowed in an event procedure, you can invoke the ThisRule's PostEvent method (see Data Types in Java) to return control to the rule at the statement following the converse.

HPS_EVENT_VIEW Method

The HPS_EVENT_VIEW method is supported only for consistency in handling events for AppBuilder-supplied window controls.

When an event is triggered, the following information is returned in HPS_EVENT_VIEW:

- *EVENT_SOURCE* - the HPSID of the control
- *EVENT_QUALIFIER* - the name of the event
- *EVENT_PARAM* - the parameters returned by the event, converted to a character string, and separated by commas

Event Parameters

Many events include parameters. To see what parameters are returned to an event procedure when the event is triggered, refer to the *ObjectSpeak Reference Guide.*

You can use DCL statements to declare the same procedure for multiple events, objects, or object types (see Event Procedure Declaration).

If you define one event procedure for a type of object and another event procedure for a particular object of that type, the procedure with the narrowest scope is invoked — the procedure for the particular object. For example, if Procedure X handles the click event for any push button, and Procedure Y handles the click event for push button Z, then if a user clicks push button Z, only Procedure Y is invoked.

The following two examples show how to define and handle events for specific objects.

Examples: Setting Event Parameters

Example 1: Defining a Particular Event of a Particular Object

The following procedure can be defined to handle the Initialize event of a Window control named MY_WINDOW:

```
PROC windowInitialize FOR Initialize OBJECT MY_WINDOW (p OBJECT TYPE InitializeEvent)
...
ENDPROC
```

Example 2: Handling a Particular Event of a Type of Object

The following procedure can be defined to handle the Initialize event for any Window control:

```
PROC windowInitialize FOR Initialize TYPE WINDOW OF GUI_KERNEL (p OBJECT TYPE InitializeEvent)
...
...
ENDPROC
```

Control Statements

This chapter describes the following types of control statements:

- Comment Statement
- ObjectSpeak Statement
- File (Database) Access Statements
- Post Event Statement
- Compiler Pragmatic Statements

Control Statement Syntax

control_statement:

comment_statement

method_invocation_statement

file_access_statement

post_event_statement

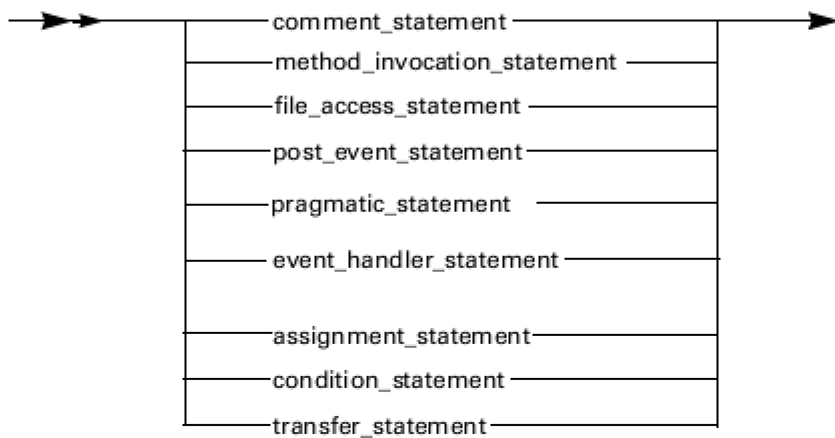
pragmatic_statement

event_handler_statement

assignment_statement

condition_statement

transfer_statement



Comment Statement

A comment describes the purpose of a Rules Language statement and is useful for other developers who look at the source code. ***>** (an asterisk and a greater-than sign) denotes the beginning and **<*** (a less-than sign and an asterisk) denotes the end of a multiline comment. Any text within these delimiters is ignored when the rule is prepared. Comments might continue across more than one line, but you cannot nest comments.

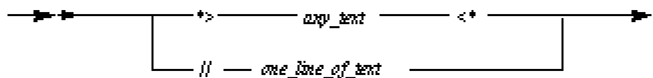
// (double forward slash) denotes single line comments.

Comment Syntax

comment_statement:

*> *any_text* <*

// *one_line_of_text*



where:

- *any_text* is any possible character sequence, including line breaks.
- *one_line_of_text* is any character sequence limited to one line without any line breaks.

Examples: Comment Statements

Examples 1 and 2 are valid comment statements, and examples 3 and 4 are invalid comment statements.

Example 1: Valid Comment Statement

```
*> assign the value 3.14 to the variable PI <*

MAP 3.14 TO PI
```

Example 2: Valid Comment Statement

```
*> This rule was last modified on January 26, 1995 <*

MAP 3.14 TO PI // Assign the value 3.14 to the variable PI
```

Example 3: Invalid Comment Statement

```
*> This rule was last modified *> by adding line 5 <* on January 26, 1995 <*
```

The compiler closes the comment after " *line 5* ". The remainder of the comment then causes a syntax error.

Example 4: Invalid Comment Statement

```
// This rule was last modified on
January 26, 1995
```

The compiler does not recognize this comment because it is on more than one line. Use `*> <*` instead for multi-line comments.

ObjectSpeak Statement

The ObjectSpeak extension to the Rules Language invokes methods for objects supplied with AppBuilder.

For a list of available objects, methods, and properties, refer to the *ObjectSpeak Reference Guide*.

ObjectSpeak Syntax

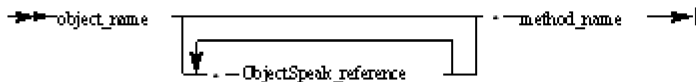
objectspeak_statement:

object_name [. objectspeak_reference (. objectspeak_reference)*] . method_name

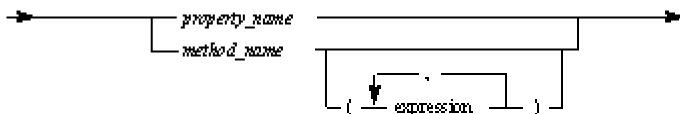
objectspeak_reference:

property_name

method_name ['(' (expression)* ')']



where ObjectSpeak_reference is:



where object_name can be:

- The system identifier (HPSID) of the object.
- The alias of the object---see Alias for information about alias of an object.
- An object---see Object Data Types for information about an object.
- An array---see Array Object for information about an array.

where:

- expression---see Expression Syntax.

Referencing Properties

You can use ObjectSpeak expressions to reference an object's properties. The simplest expression is the name of an object, followed by a period, followed by the name of the property. You can use such an expression in any statement in which an ordinary expression can be used. For example:

```
MAP 12 TO myAnimatedButton.TextXPos

MAP myAnimatedButton.Speed TO saveSpeed
```

Nested Properties

Sometimes a property of a control is "buried" inside the control. For example, a property that is returned by a method of the control or a property of a property of the control are nested. In theory, the nesting can go to any level. When a property is nested, simply use a period to separate each "nesting level." For example:

```
myGauge.Picture.Handle
```

In this example, `Handle` is a property of the `Picture` property of the Gauge control named `myGauge`.

When a property is deeply nested inside a control, you can use an object to abbreviate the property reference. For example, suppose there is a property named `Text` that is nested in this way (where `method2` returns an object of type BitMapButton):

```
Object_name.method1(parm,parm).method2(parm).Text
```

If you declare an object as follows (where `theBitMapButton` is an arbitrary name):

```
DCL theBitMapButton OBJECT TYPE BitMapButton of GUI_KERNEL ENDDCL
```

then you can abbreviate references to the `Text` property, as follows:

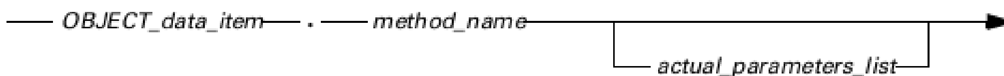
```
theBitMapButton.Text
```

Object Method Call

Object Method Call Syntax

object_method_call:

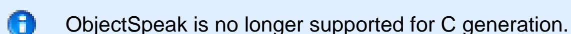
object_data_item . *method_name* [*actual_parameters_list*]



where:

- *actual_parameters_list* is a list of actual parameters delimited with commas and enclosed in parentheses. If the list is empty, parentheses can be omitted. If a method does not have parameters, empty parentheses (()) can be written. For more information, see:
- Object Method Call in Java

For information about the conversion between Java standard data types and Rules Language data types when passing parameters to and accepting return values from Java methods, refer to ObjectSpeak Conversions in Java.


ObjectSpeak is no longer supported for C generation.

Invoking Methods for Objects

The simplest method invocation is the object name followed by a period, and then the method name.

Do *not* include parentheses if the method has no parameters

For example:

```
CommonDialog.ShowHelp
```

If the method takes parameters, they are enclosed within parentheses following the method name, and separated with commas. For example:

```
treeView.HitTest(100,200)
```

Invoking Nested Methods

Sometimes a method of a control is "buried" inside the control; for example, it might be a method of a property that is returned by a method of the control. In theory, the nesting can go to any level. When a method is nested, simply use a period to separate each "nesting level." For example:

```
myWindow.Size.setWidth(newWidth)
```

The `setWidth` method is a method of the `Size` property of the Window control named `myWindow`.

File (Database) Access Statements

The following statements are used for file access (or database access):

- SQL ASIS Support
- START TRANSACTION
- COMMIT TRANSACTION
- ROLLBACK TRANSACTION

For information about Java development, refer to Transaction Support in Java.

File (Database) Access Syntax

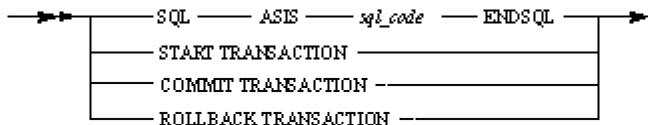
file_access_statement:

SQL ASIS *sql_code* ENDSQL

START TRANSACTION

COMMIT TRANSACTION

ROLLBACK TRANSACTION



SQL ASIS Support

The Rules Language supports access to various databases using SQL code embedded directly in a rule. A rule accesses a database directly by executing embedded SQL code specified as an argument to the SQL ASIS statement. All literals in the SQL ASIS statements must be written using the DBMS notation, not the Rules Language syntax. SQL DBMSs also distinguish between single and double quotes, but they have different meanings.

Literals in SQL code are not parsed, but are passed to the DBMS unchanged. So, during preparation, in order to avoid semantic confusions, single-quoted strings are SQL character literals, and double-quoted ones are quoted SQL identifiers.

DB double quoted identifiers are often used when naming DB objects with keywords (e.g. table "column" with column "table" in CREATE TABLE "COLUMN" ("TABLE" integer)). Therefore, all code within SQL ASIS block must follow SQL syntax, not Rules Language syntax. For example, you cannot use the double quotes instead of the single quotes to define a string literal.

Example: Correct and Incorrect SQL code

1. Correct SQL code sample

```
SQL ASIS

select * into curl
from customer
where customer.firstname = :custfname and
     "customer"."lastname" = 'Jones'

ENDSQL
```

2. Incorrect SQL code sample
(Double-quoted "Jones" can be used as character literal in the rule, but not within SQL ASIS block where it will be treated as DB identifier):

```
SQL ASIS

select * into curl
from customer
where customer.firstname = :custfname and
     "customer"."lastname" = " Jones "

ENDSQL
```

Because it passes the embedded SQL directly to the underlying database's SQL compiler, the AppBuilder environment supports whatever ANSI version the underlying database supports. Thus, you can use non-standard extensions supported by your database. However, doing so might cause problems when porting the code to another database.

Refer to SQL ASIS Support in Java for special considerations in Java.

Usage

SQL statements cannot be nested. The code between the keywords SQL ASIS and ENDSQL is copied "as is" into the generated code. The following are exceptions to this:

- Host variables
- Comments
- SQL Communication Area (SQLCA)

Most SQL ASIS statements reference database tables that are created by right-clicking the File object in the Hierarchy and selecting Prepare, or by selecting Prepare from the Build menu. For these statements, you must use the implementation names of the files and fields used to create the table. But host variables (expressions beginning with a colon) correspond to variables in your rule, not to any database object, so they must be coded using the long name.
The SQL code can use any non-multiple-occurring view or field in the rule's data universe or any locally-declared variable as a host variable.

A field name must be clearly specified. In SQL code within a rule, qualify a field name by writing the view name first, and then the field name, separating them by a period rather than by the keyword OF. (This is similar to the syntax used in PL/I or C to identify the components of a structured variable.) Thus, a field that might appear elsewhere in a rule as LAST_NAME OF CUSTOMER must appear in embedded SQL code as:

```
:CUSTOMER.LAST_NAME
```

SQL supports only one level of qualification. That is, you must redefine the view to "flatten" it. To access a structure that requires multiple levels of qualification, declare a local variable LIKE your target structure and reference the variable in the SQL code. You can then map data from your variable to the "real" data structure.

Because SQL code cannot access multiple-occurring views, you cannot simply SELECT data into an AppBuilder array. However, by declaring an SQL cursor into a selection, you can loop through the selection so that each iteration of the loop can FETCH the row under the cursor INTO a locally declared view acting as a host variable. Then you can MAP that view into one occurrence of a multiple-occurring view.

In order for the loop to know when it has fetched all of the rows that the SELECT returns, SQLCODE and the rest of the SQL Communication Area's variables are accessible to any rule that uses the SQL ASIS command, just as if they were locally declared fields. You can view the SQLCA in the RuleView to examine the SQL return codes. And you can use the SQL return code values in your rule.

The SQLCA view is automatically appended to any rule that has DB2 usage, so you do not need to attach it to the rule yourself.

Do not check the SQLCODE field after a DCL CURSOR, INCLUDE, or WHENEVER statement because such statements do not affect the SQLCA.

Do not use `SELECT *` in embedded SQL; enhancements to the table structure might make your host variables incompatible with the rows of the table.

Do not use *<:* or *:>* in embedded SQL statements because these special symbols are used for suppressing macro substitution for strings (see Using Quoted Strings in Macros). However, you can use < *:* or *:* > with a space between < (or >) and *:* symbols, and a variable.

You must write and invoke a user component to access any file that does not support SQL statements. Refer to *Developing Applications Guide* for instructions for writing a user component.

Using SQL host variables in the rules for Classic COBOL, OpenCOBOL, and C generation.

During code generations, only limited analysis of the SQL code within SQL ASIS ENDSQL blocks is performed. Mainly, the host variables used in the code are analyzed. The rest of the code is generated in the COBOL program AS IS, without changes.

ClassicCOBOL DATE, TIME, TIMESTAMP, and DEC fields do not have the same representation as corresponding DB2 column types. To solve this problem, all host variables of these types are converted to DB2 representation. The converted value is stored in the temporary variable, and this temporary variable is used as a host variable in the generated COBOL code. Values are converted before and after each SQL ASIS block. All host variables used in the block are converted to DB2 types before SQL block. All host variables are converted back to COBOL representation after SQL block is executed. No analysis is performed to determine which variable is input variable and which one is output.
In OpenCOBOL no conversion is necessary because data types used in COBOL programs are the same as DB2 with the exception of TIME fields. This conversion problem is solved using REDEFINE fields.

DB2 does not allow DATE value 0000/00/00, which is the initial value for DATE fields in AppBuilder. Values with year 0 are not valid values in DB2; however, in AppBuilder they are valid values. To solve this problem, each host variable of DATE type containing the value less than 367 (year 0 in AppBuilder) is replaced with the value 367 or 0001/01/01, then converted to a DB2 representation. After each SQL block, all data values that are less than or equal to 367 are replaced with value 0. This verification is done in ClassicCOBOL and OpenCOBOL. OpenCOBOL has an option to use string comparison instead of converting the date to a number and comparing it with a value 367.

Examples: SQL ASIS Statements

Example 1: Using SQL ASIS with host variables

```
DCL
  myDate date;
ENDDCL

SQL ASIS
  Declare myCursor cursor for
  Select
    Column1,
    Column2
  From myTable
  Where Column3 = :myDate
ENDSQL

SQL ASIS
  open myCursor
ENDSQL
```

If we generate ClassicCOBOL code for that, it will be as follows (simplified for clarity):

```
* SQL ASIS
* CONVERSIONS BEFORE SQL
* DATE TO CHAR
IF (V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS < 367) THEN MOVE
 367 TO V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS.
CALL 'CGDT2CH' USING DFHEIBLK DFHCOMMAREA V-SMT-SQL-1, V--
LOC-DATE-0001 OF
V-SQLRULE-LOCAL-VARS

EXEC SQL
  Declare myCursor cursor for
  Select
    Column1,
    Column2
  From myTable
  Where Column3 = :V--SMT-SQL-1
END-EXEC

* CONVERSIONS AFTER SQL
* CHAR TO DATE
CALL 'CGCH2DT' USING DFHEIBLK DFHCOMMAREA V--LOC-DATE-0001 OF
V-SQLRULE-LOCAL-VARS, V-SMT-SQL-1
IF (V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS < 368) THEN MOVE
 0 TO V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS.

EXEC SQL
  open myCursor
END-EXEC
```

This code works because the value of `V--SMT-SQL-1`, which is implicitly used in OPEN statement has not changed between EXEC SQL statements. However, if we change the original rule code to be:

```
DCL
  myDate date;
ENDDCL

SQL ASIS
  Declare myCursor cursor for
  Select
    Column1,
    Column2
  From myTable
  Where Column3 = :myDate
ENDSQL

set myDate := date ()

SQL ASIS
  open myCursor
ENDSQL
```

Then COBOL code will be:

```
* SQL ASIS
* CONVERSIONS BEFORE SQL
* DATE TO CHAR
IF (V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS < 367) THEN MOVE
 367 TO V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS.
CALL 'CGDT2CH' USING DFHEIBLK DFHCOMMAREA V--SMT-SQL-1, V--LOC-
DATE-0001 OF V-SQLRULE-LOCAL-VARS

EXEC SQL
  Declare myCursor cursor for
  Select
    Column1,
    Column2
  From myTable
  Where Column3 = :V--SMT-SQL-1
END-EXEC

* CONVERSIONS AFTER SQL
* CHAR TO DATE
CALL 'CGCH2DT' USING DFHEIBLK DFHCOMMAREA V--LOC-DATE-0001 OF
V-SQLRULE-LOCAL-VARS, V--SMT-SQL-1
IF (V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS < 368) THEN MOVE
 0 TO V--LOC-DATE-0001 OF V-SQLRULE-LOCAL-VARS.

CALL 'CGDATE' USING DFHEIBLK, DFHCOMMAREA, TMP0 OF
V-SQLRULE-TEMP-VARS.
MOVE TMP0 OF V-SQLRULE-TEMP-VARS TO V--LOC-DATE-0001 OF
V-SQLRULE-LOCAL-VARS.

EXEC SQL
  open myCursor
END-EXEC
```

This code will produce the wrong result, because `V-LOC-DATE-0001` has changed, but `V-SMT-SQL-1` has not been updated. This happened because OPEN statement has no host variables and there is no conversion.
To solve this problem the code generation option -H was introduced. When it is used, all host variables used in any SQL ASIS block in the rule are converted and verified before and after each SQL block. This might be unnecessary overhead, but it is the only way to solve the problem without implementing the SQL parser.

Example 2: SQL Sample
The following rule reads all the customers with a given last name from the `CUSTOMER_TABLE` database into a multiple-occurring view called `CUSTOMER_LIST`. The name to look for is in the field `SEARCH_NAME` of the rule's input view. The SELECT statement lists the DB2 column names equivalent to the fields of interest.

```
DCL
  I INTEGER;
  CUSTOMER_TEMP LIKE CUSTOMER;
ENDDCL

SQL ASIS
  DECLARE CURS_1 CURSOR FOR
  SELECT LAST_NAME, FIRST_NAME, ID_NUM
  FROM CUSTOMER_TABLE
  WHERE LAST_NAME = :RULE3I.SEARCH_NAME
ENDSQL

DO FROM 1 TO 25 INDEX I
  SQL ASIS
    FETCH CURS_1 INTO :CUSTOMER_TEMP
  ENDSQL
WHILE SQLCODE = 0
  MAP CUSTOMER_TEMP TO CUSTOMER_LIST OF RULE3O (I)
ENDDO
```

START TRANSACTION

Use START TRANSACTION to explicitly start a database transaction. A transaction is started implicitly when you use a remote rule (one running on a server machine) that accesses a database.

If a transaction is started explicitly, you can commit or roll back changes to both local and remote databases subsequent to the START TRANSACTION using COMMIT TRANSACTION or ROLLBACK TRANSACTION.

If a transaction is started implicitly, changes to remote databases are committed or rolled back depending upon settings in the DNA.INI file. Unless a transaction is committed explicitly using COMMIT TRANSACTION, changes to local databases are not committed until the AppBuilder process terminates.

COMMIT TRANSACTION

Use COMMIT TRANSACTION to commit changes to local and remote databases since the previous START TRANSACTION statement. Although you can also commit a database using SQL ASIS COMMIT, the advantages of using COMMIT TRANSACTION include the following:

- It is database independent. (AppBuilder translates it for the database being used.)
- It works on local databases as well as remote databases. By contrast, implicit commits affect only remote databases on server machines.

The commits performed by a COMMIT TRANSACTION are not coordinated among multiple locations. For example, a remote database commit could succeed while a local one fails.

ROLLBACK TRANSACTION

Use ROLLBACK TRANSACTION to roll back changes to local and remote databases since the previous START TRANSACTION statement. Although you can also roll back a database using SQL ASIS ROLLBACK, the advantages of using ROLLBACK TRANSACTION include the following:

- It is database independent. (AppBuilder translates it for the database being used.)
- It works on local databases as well as remote databases. By contrast, implicit rollbacks affect only remote databases on server machines.

Post Event Statement

Use a POST EVENT statement to post a message to another application or to a different rule in the same application. The message text is contained in the view attached to the event. See CONVERSE for Global Eventing for information about posting and receiving global-event messages.

POST EVENT Syntax

post_event_statement:

POST EVENT *event_name* (*event_name*)*

For example:

```
POST EVENT CUTOFF_REACHED START_NEW
```

Compiler Pragmatic Statements

Compiler PRAGMA statements are special commands that control certain features of the compiler.

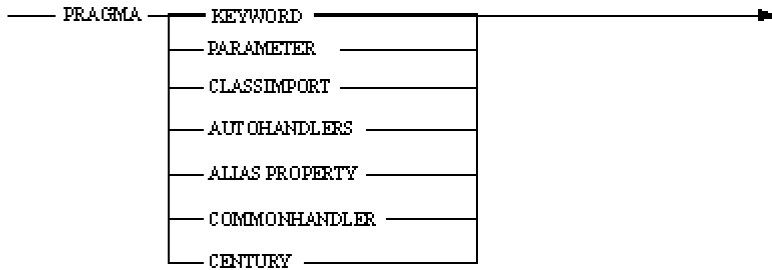
PRAGMA Syntax

pragmatic_statement:

PRAGMA pragma_name

pragma_name:

one of KEYWORD PARAMETER CLASSIMPORT AUTOHANDLERS ALIAS PROPERTY COMMONHANDLER CENTURY



The compiler PRAGMA statement can be used between any language constructions but cannot be used inside of a construction, such as IF, CASEOF, DO, or PROC statements. The PRAGMA statement affects subsequent statements within the same rule code, therefore, the PRAGMA statement should not be the last statement in the rule.
This section describes the PRAGMA KEYWORD and PRAGMA PARAMETER statement. For other PRAGMA statements, refer to the following:

- PRAGMA CLASSIMPORT in Java
- PRAGMA AUTOHANDLERS in Java
- PRAGMA ALIAS PROPERTY in Java
- PRAGMA COMMONHANDLER in Java
- PRAGMA CENTURY for OpenCOBOL

PRAGMA KEYWORD

Use the PRAGMA KEYWORD to switch selected Rules Language keywords on or off. The default value is ON for all of the keywords listed.

PRAGMA KEYWORD Syntax

pragma_keyword_statement:

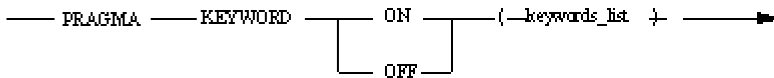
PRAGMA KEYWORD keyword_switcher '(' keyword_list ')'

keyword_switcher:

one of ON OFF

keyword_list:

keyword (, keyword)*



where:

- keywords_list is the parameters list of keywords to switch on or off. Separate individual keywords using commas (spaces are ignored) and place the entire list in parentheses. The PRAGMA KEYWORD clause is *not* case-sensitive, so keywords can be lower or uppercase.

> If the PRAGMA KEYWORD OFF (PRAGMA) clause is used, it must be the last PRAGMA statement in a rule.

Not all keywords can be turned on or off. The Rules Language keywords that can be switched on or off with the PRAGMA clause are listed in the following sections:

- Keywords for Java that can be Disabled with PRAGMA KEYWORD
- Keywords for C that can be Disabled with PRAGMA KEYWORD
- Keywords for ClassicCOBOL that can be Disabled with PRAGMA KEYWORD
- Keywords for OpenCOBOL that can be Disabled with PRAGMA KEYWORD,

Example: Using PRAGMA to Switch Keywords On and Off

Keywords true and false are switched on and off.

```
PRAGMA KEYWORD on (true)
PRAGMA KEYWORD off (true)
PRAGMA KEYWORD on (true,false)
PRAGMA KEYWORD off (false,true)
```

PRAGMA PARAMETER

PRAGMA PARAMETER Syntax

pragma_parameter_statement:

PRAGMA PARAMETER '(' *parameter_name*, *parameter_value* ')'

where:

- *parameter_name* – is a name of some codegen parameter or keyword FLAG
- *parameter_value* – is a new value of this parameter or the corresponding flag.

This pragma allows you to overwrite or specify new parameters or codegen flags from [CodegenParameters] section of HPS.INI file directly from the rule code. These parameters are effective only for the rule where PRAGMA statement is used.
For example, the following statement

```
PRAGMA PARAMETER (JAVA_PERSISTENT_CURSOR, YES)
```

enables persistent cursor generation.

In the following example, the statement

```
PRAGMA PARAMETER (FLAG, "RTCALL")
```

determines the generation of all Date/Time function calls as runtime calls (for Open Cobol generation).
The restriction is that PRAGMA PARAMETER statements are processed during the rule code parsing stage, thus they, for instance, are not affecting the bind file parsing stage or panel file parsing. Also some parameters, requiring additional configuration before rule code parsing, could not be affected by this PRAGMA.
There is a full list of all flags and parameters which are allowed to be used inside PRAGMA PARAMETER clause:

- flags:
    - A
    - I
    - OVE
    - MEXCI
    - DYNCALL
    - CUSTCALL
    - MOVEC
    - GENPERIOD
    - GENNOSUFF
    - RTCALL
    - NEWDT
    - VERDT
    - RTDTI
    - NCOCC
    - VCTRACE
    - ROCRS
    - NATIONAL
- parameters:
    - CompareDatesAsString
    - OCC_VIEW_SIZE_THRESHOLD
    - INLINE_VIEW_COPY
    - INLINE_VIEW_COPY_FIELDS_LIMIT

- COPYFROM_NULL_PARAMETERS_THRESHOLD
- LAZY_INSTANTIATION_ENABLED
- INITIALIZE_VIEW
- GENERATE_NO_SUFFIX
- CHECK_DEC_FORMAT.

Assignment Statements

Assignment statements allow you to assign a new value to a variable data item. These statements include:

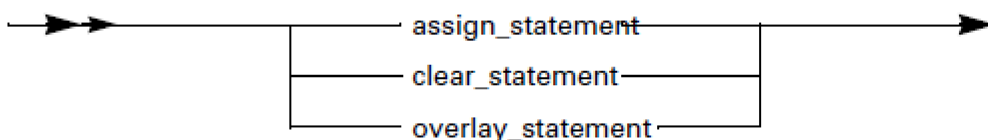
- Assignment Statements
- CLEAR Statement
- OVERLAY Statement

Besides using the three statements listed above, you can also assign new data to a view by using the special redefine capability. You can also assign variables of the Object data type. These alternatives are discussed in the following sections:

- Redefining Views
- Assigning Object Data Type Variables in Java

Assignment Statement Syntax

```
assignment_statement:
      one of assign_statement clear_statement overlay_statement

assign_statement:
      one of map_statement set_statement

map_statement:
      MAP expression TO variable_data_item ( , variable_data_item )*
      MAP source_for_view TO view ( , view )*

set_statement:
      SET variable_data_item ( , variable_data_item )* set_operator expression
      SET view ( , view )* := source_for_view

source_for_view:
      view
      aggregate
      rule_name [ '(' rule_comp_parameter ')' ]
      component_name [ '(' rule_comp_parameter ')' ]

rule_comp_parameter:
      view
      aggregate
      expression ( , expression )*

set_operator:
      one of := +:= -:=
```

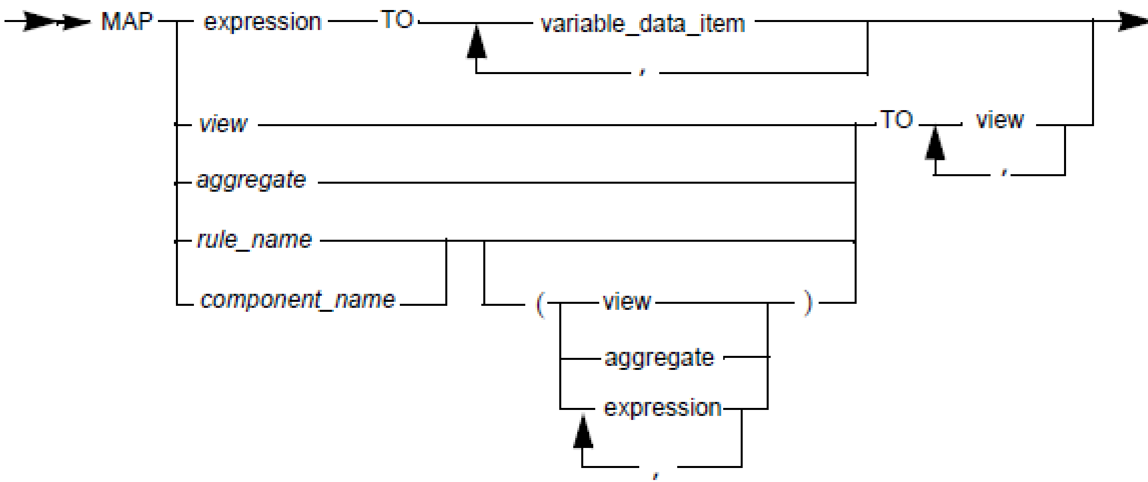


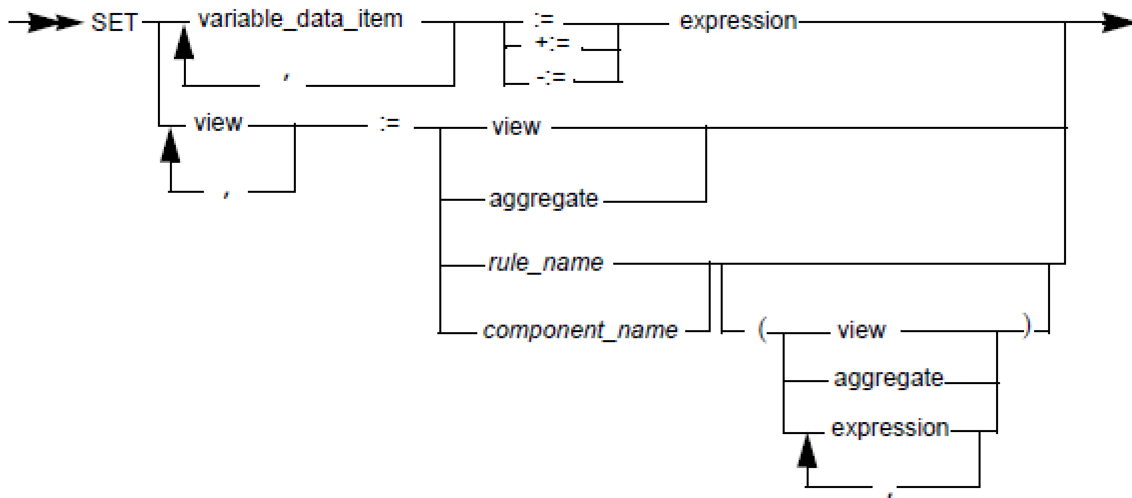
Assignment Statements

A MAP statement copies the value of the source item to the target item. You can map any valid expression to a field variable, provided that the expression and the field are of compatible data types. Refer to the following related topics:

- Increment and Decrement SET Statements
- Using Aggregates
- Data Type Mapping Errors
- Mapping Data

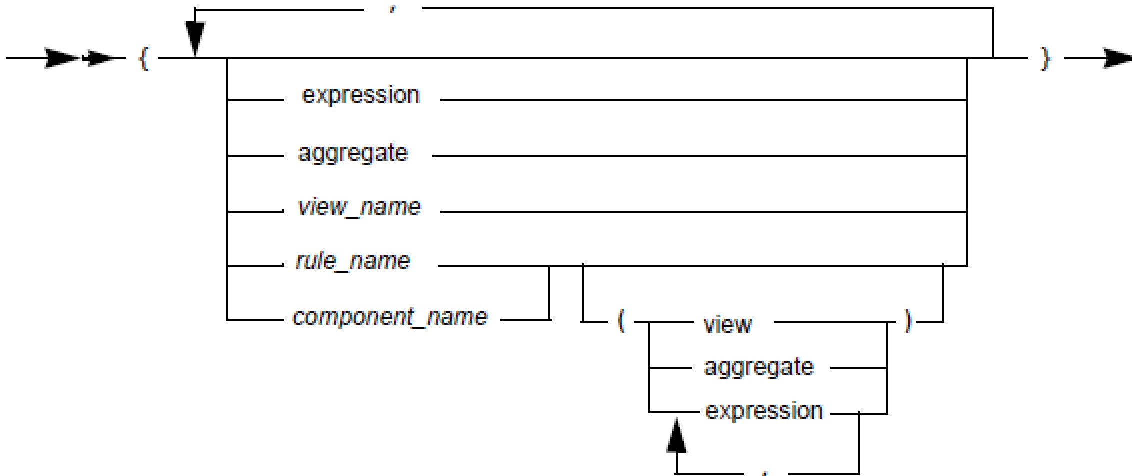
MAP and SET Syntax





Aggregate Syntax

```
aggregate:
       { ( aggregate_element )* }

aggregate_element:
       expression
       aggregate
       view_name
       rule_name [ '(' rule_comp_parameter ')' ]
       component_name [ '(' rule_comp_parameter ')' ]


rule_comp_parameter:
       view
       aggregate
       expression ( , expression )*
```



where:

- expression – see Numeric Expressions.
- variable_data_item – see Variable Data Item.
- view – see View.

Usage

If the source item is a numeric expression and the target item is multiple data items (separated by a comma), the value of the source expression is calculated separately before each mapping. This allows for a slightly different value to be mapped to each target data item depending upon the type (precision) of the target item.

If the variable is a view, you can only map another view to it. This maps the data in any field under the first view to a field with the same name under the second view. In addition, it maps data from a field in a subview of the first view to a field in the subview of the second view under the following circumstances:

- Subviews are directly attached to the views being mapped.
- Names of the subviews are identical.

You can also map to or from a multiple-occurring view. If both views are multiple-occurring, this maps the same numbered view from the first view to the other as for a regular view. Thus,

```
MAP VIEW_A TO VIEW_B
```

maps `VIEW_A(1)` to `VIEW_B(1)`, `VIEW_A(2)` to `VIEW_B(2)`, and so on, until one of the two views runs out of occurrences. If only one of the views is multiple-occurring, the data in the non-occurring view maps to or from the first occurring view. Thus, in the statement above, if `VIEW_A` is not multiple-occurring but `VIEW_B` is, the statement maps `VIEW_A` to `VIEW_B(1)`. Conversely, if `VIEW_A` is multiple-occurring and `VIEW_B` is not, the statement maps the first occurrence of `VIEW_A` to `VIEW_B`.

The source of a MAP statement can be a RULE call; that output view is mapped to the destination after the rule call. In this case, the rule name should be used without USE. For example:

```
MAP integer_sum (1, 32500) to V
MAP res of v TO f3
```

The source of the MAP can be any method call or any variable from the object. If the object is a window object, then its system identifier (HPSID) can be used without declaring this object in the rule's DCL section.

Assume there is a window attached to the rule, and this window contains the object PUSHBUTTON with the system identifier (HPSID) of 'OK_BUTTON'. Use the following syntax in Java:

```
MAP OK_Button.Foreground to OK_Button.Background
MAP 'OK' TO OK_Button.Text
```

and in C, use:

```
MAP OK_Button.ForeColor to color
MAP OK_Button.Text TO ButtonText
```

The SET statement is an analog of the MAP statement with certain limitations, which can be seen on the syntax diagram.

Increment and Decrement SET Statements

A SET statement can be used to increment or decrement a numeric variable by any value without writing addition or subtraction. To do this, write +:= or -:= . The right side expression will be added or subtracted from the *variable_data_item* . For example:

```
SET I:=1
SET I+:=1 *>sets I to 2<*
SET I-:=1 *>sets I to 1<*
```

Using Aggregates

The source of MAP can be aggregate. Using an aggregate allows you to map several values with a single map statement.

Example: Using aggregate in MAP statement

```
DCL
  last_name, first_name varchar (20);
  birthday date;
  age integer;
  age_view view contains birthday, age;
  person_view view contains last_name, first_name, age_view;
  my_age like age_view;
ENDDCL
```

The following example uses an aggregate to map three values to person_view:

```
MAP {"last name", "first name", my_age } to person_view
```

The following example uses nested aggregates to map to person_view:

```
MAP {"last name", "first name", {DATE('12/29/61','%m/%d/%y'),34}} to person_view
```

Data Type Mapping Errors

Any attempt to map a field or constant to a field of an incompatible data type produces an error when you try to prepare the rule. In addition, you get a warning message for any MAP statement whose source field data type is potentially incompatible with the data type of its destination field. These messages flag statements that might, under certain conditions, lead to errors or unpredictable results at runtime.

For example, it is perfectly legal to map an INTEGER field to a SMALLINT field because the two data types are compatible. However, when the rule is executed, any INTEGER field containing a value greater than 32,767 or less than -32,767 maps incorrectly to the SMALLINT field. For example, if the source field equals +32,768, the target field becomes -1. If you have a rule that maps an INTEGER field to a SMALLINT field, a warning message is added to your preparation results file.

Mapping Data

Mapping of data are described in the following sections:

- Mapping To and From a PIC Field
- Mapping To a DEC Field

- Mapping Between Fields of Different Lengths
- Mapping To and From a VARCHAR Field
- Mapping To and From a TEXT or IMAGE Field
- Mapping To or From a DBCS or MIXED Field
- Mapping a View to a View

Example: Mapping Data to a Field

The following rules code illustrates the most common form of mapping data to a field:

```
MAP 10 TO NUMBER_OF_PEOPLE
*> Copies the value of the numeric literal 10 into the field <*

MAP '223 West 21st Street' TO ADDRESS OF CUSTOMER_DETAIL
*> Copies the value of the character literal <*
*> '223 West 21st Street' into the field ADDRESS <*

MAP ADDRESS OF CUSTOMER_DETAIL TO ADDRESS OF SHOW_CUSTOMER_DETAIL
*> Copies the value of the first ADDRESS field into the second <*
*> ADDRESS field <*

MAP (PRICE - DISCOUNT) * TAX_RATE TO SALES_TAX
*> Copies the value to which the expression resolves <*
*> into the field SALES_TAX <*

MAP JAN IN MONTHSET TO WHICH_MONTH
*> Copies the value of the symbol Jan [not the symbol name] <*
*> into the field WHICH_MONTH <*
```

Mapping To and From a PIC Field

You can map an unsigned PIC field to either a character or a numeric field, but you cannot map a character value to any PIC field. You cannot map a PIC field containing either a sign code ('S') or a decimal placeholder ('V') to a CHAR or VARCHAR field. When mapping a valid PIC field to a CHAR or VARCHAR field, the value of the character field is set to a string representing the number in the PIC field. Any string longer than the character field to which it is mapped is truncated.

A warning is issued in the preparation results if you map a field of format SMALLINT, INTEGER, or DECIMAL to a PIC field whose picture does not begin with an 'S' to allow for a negative sign.

Mapping To a DEC Field

Mapping more digits than allowed to a DEC field results in a runtime error, causing RuleView to display asterisks in the field.

Mapping Between Fields of Different Lengths

Although the data types are compatible, an error is issued if you attempt to map a numeric constant to a DECIMAL or PIC field that either does not have enough places to the left of the decimal to hold its integer part, or does not have enough places to the right of the decimal to hold its fractional part.

An error is not issued for Java generation in only one case, when the source value is a set symbol. This is needed in order to support separate generation of sets and rules.

Similarly, a CHAR or VARCHAR destination field might be too short to contain a source field that is mapped to it. In this case, the destination field stores only as many characters as can fit from the start of the string. Conversely, a string is left justified if the length of its destination field is greater than the length of the source field. The remaining positions are filled with blanks.

Mapping To and From a VARCHAR Field

Mapping to a VARCHAR field is a complicated procedure because a VARCHAR field contains a length property. When a string is mapped into a VARCHAR field, the length associated with that VARCHAR field is set to the length of the string, and the remaining character spaces in the VARCHAR field are set to blanks. For example, assume the length of a VARCHAR field is 5, and the field contains the string *'Hello'* . If it is mapped to a VARCHAR field with a maximum length of 10, then the destination field is set to *'Hello* ', and its length is set to 5.

If a source string is longer than the maximum length of a target VARCHAR field, the characters that fit into the field are mapped and the length is set to the maximum length of the field. For example, if the string *'Hello* ' is mapped to a VARCHAR field with a maximum length of 3, the destination field is set to *'Hel',* and its length is set to 3. If a source string is shorter than a target VARCHAR field, any positions in the target not containing new data are set to blanks, and the length of the target is set to the length of the source.

When a CHAR field is mapped to a VARCHAR field, the length associated with the VARCHAR field is set to the defined length of the CHAR field, even if the string in the CHAR field is shorter. Thus, if the string *'Hello'* is stored in a CHAR (10) field and then mapped to a VARCHAR (20) field, the length of the VARCHAR field is set to the length of the CHAR field (10) and not to the length of the string (5).

In summary, assume B is a variable of type VARCHAR and A is a character value (a variable, a literal of type VARCHAR or type CHAR, or a symbol of type CHAR). The length of B is determined from the length of A in a MAP A TO B statement as follows:

- If length of A <= maximum length of B, then Length of B = length of A. The contents of B equals the contents of A padded with spaces to the right.
- If length of A > maximum length of B, then Length of B = maximum length of B. The contents of B equals the first "max length of B" characters of A.

Example: Mapping Data to a VARCHAR Field

The following Rules code example illustrates how mapping data to a VARCHAR field affects its contents and length:

```
DCL
  CHAR_VAR_1 CHAR (10);
  CHAR_VAR_2 CHAR (20);
  VARCH_VAR_1 VARCHAR (15);
  VARCH_VAR_2 VARCHAR (20);
ENDDCL

MAP 'ABC ' TO CHAR_VAR_1
MAP '* MY LENGTH IS 20 *' TO CHAR_VAR_2
MAP 'ABC ' TO VARCH_VAR_1
*> Copies the value 'ABC ' into varch_var_1 and <*
*> sets the length of varch_var_1 to 4, the length of 'ABC ' <*

MAP CHAR_VAR_1 TO VARCH_VAR_1
*> Copies the value 'ABC ' into varch_var_1 and <*
*> sets length of varch_var_1 to 10, the length of char_var_1 <*

MAP CHAR_VAR_2 TO VARCH_VAR_1
*> Copies the value '* MY LENGTH IS' into varch_var_1 and <*
*> sets length of varch_var_1 to 15, the length of VARCH_VAR_1 <*

MAP CHAR_VAR_2 TO VARCH_VAR_2
*> Copies the value '* MY LENGTH IS 20 *' into varch_var_2 and<*
*> sets length of varch_var_2 to 20, the length of VARCH_VAR_2 <*

MAP VARCH_VAR_2 TO VARCH_VAR_1
*> Copies the value '* MY LENGTH IS' into varch_var_1 and <*
*> sets length of varch_var_1 to 15, the length of VARCH_VAR_1 <*

MAP VARCH_VAR_1 TO VARCH_VAR_2
*> Copies the value '* MY LENGTH IS ' into varch_var_2 and <*
*> sets length of varch_var_2 to 15, the length of VARCH_VAR_1 <*
```

Mapping To and From a TEXT or IMAGE Field

Fields of these types are stored as CHAR (256) fields, so the conditions that apply to CHAR fields also apply to TEXT and IMAGE fields. In addition, mapping is the only operation you can perform on a TEXT or IMAGE field. That is, you can map a value between two TEXT fields or between a TEXT field and a character field. Similarly, you can map a value between two IMAGE fields or between an IMAGE field and a character field. You cannot map a value from a TEXT field to an IMAGE field, nor can you map a value from an IMAGE field to a TEXT field.

Example: Mapping to and from a TEXT or IMAGE Field

Assume that a rule contains the following statements:

```
DCL
  LOGO_FILE IMAGE;
  INFO_FILE_1 TEXT;
  INFO_FILE_2 TEXT;
  CHAR_FIELD CHAR (256);
ENDDCL
```

You can perform the following operations on these variables:

```
MAP 'd:\bitmaps\our_logo' TO LOGO_FILE
*> Copies the indicated string to logo_file <*

MAP LOGO_FILE TO CHAR_FIELD
*> Copies the value in logo_file to char_field <*

MAP INFO_FILE_1 TO INFO_FILE_2
*> Copies the value in info_file_1 to info_file_2 <*
```

Mapping To or From a DBCS or MIXED Field

You can map a MIXED field to another MIXED field or a DBCS field to another DBCS field. However, to map between data types where assignment is not directly allowed, you must use the appropriate conversion function, either CHAR, MIXED, or DBCS. The MIXED and DBCS data types are discussed in DBCS and MIXED Data Types, and the conversion functions are discussed in Double-Byte Character Set Functions. Standard warnings about possible truncation still apply in any situation.

Whenever a value of any acceptable type is being assigned to a MIXED or DBCS variable, it is validated.

In Java and ClassicCOBOL, validation determines whether or not the source actually is a valid MIXED or DBCS value according to the specified codepage. In OpenCOBOL, assignment does not perform such validation. It only verifies that the shift control characters are balanced for MIXED fields.
In Java, the validation codepage is specified by the DBCS_VALIDATION_CODEPAGE parameter in the [VALIDATION] section of the appbuilder.ini file. This ini setting can be changed without recompilation. If validation fails, an exception is raised at runtime.

In ClassicCOBOL, validation verifies that the shift control characters are balanced and both bytes of each DBCS character are either 0x40 (DBCS space) or in the 0x41-0xFE (inclusive) range. If validation fails, the function returns spaces and in the case of DBCS data types an error message is issued at runtime.

Example: Mapping to and from a DBCS or MIXED Field

This example applies to C, assuming that a rule contains the following statements:

```
DCL
  C1 CHAR (10);
  M1 MIXED (10);
  D1 DBCS (10);
ENDDCL
```

Assume also that the repository contains a:

- Set SET_C of type CHAR (10) with symbols SYM_C_1
- Set SET_M of type MIXED (10) with symbols SYM_M_1
- Set SET_D of type DBCS (10) with symbols SYM_D_1

In that case, you *cannot* use the following statements:

```
MAP D1 TO C1
MAP D1 TO M1
MAP M1 TO C1
MAP M1 TO D1
MAP C1 TO M1
MAP C1 TO D1
MAP SYM_D_1 TO C1
MAP SYM_D_1 TO M1
MAP SYM_M_1 TO C1
MAP SYM_M_1 TO D1
MAP SYM_C_1 TO M1
MAP SYM_C_1 TO D1
```

Instead, you *must* use these statements:

```
MAP CHAR(D1) TO C1
MAP MIXED(D1) TO M1
MAP CHAR(M1) TO C1
MAP MIXED(C1) TO M1
MAP CHAR(SYM_D_1) TO C1
MAP MIXED(SYM_D_1) TO M1
MAP CHAR(SYM_M_1) TO C1
MAP DBCS(SYM_M_1) TO D1
MAP MIXED(SYM_C_1) TO M1
MAP DBCS(SYM_C_1) TO D1
```

The analogy holds true for character literals as well:

```
MAP CHAR ('ABCDE') TO C1
MAP MIXED ('ABCDE') TO M1
MAP DBCS ('#@') TO D1
```

These statements could also be written as:

```
MAP 'ABCDE' TO C1
MAP 'ABCDE' TO M1
MAP '#@' TO D1
```

Mapping a View to a View

AppBuilder uses two methods to map one view to another. The code generation utility selects the method to use; however, it is helpful to understand how it chooses the methods and how they work.

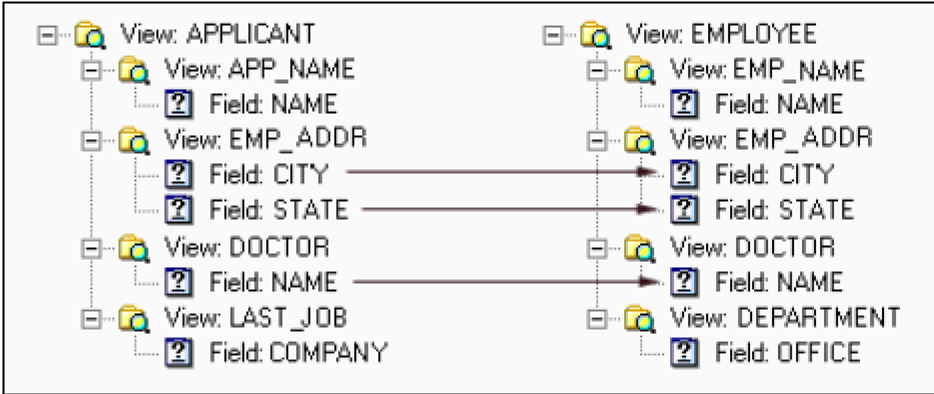
- Map Same-Named Fields
- Map Same-Typed Fields

The first one is chosen by the code generation facility when there is at least one field in the source view that has the same name as the field in the target view. If all the fields in the source and target views have different names, the second method is used.

Map Same-Named Fields

With this method, a field in the source view is copied to a field in the target view if both fields have the same name and occupy the same relative position (only have same-named parents) in both views.

For example, assume you have two views, `APPLICANT` and `EMPLOYEE`, as shown in the following figure:



Two views for MAP example

The statement

```
MAP APPLICANT TO EMPLOYEE
```

copies the data in the fields:

- CITY OF EMP_ADDR OF APPLICANT
- STATE OF EMP_ADDR OF APPLICANT
- NAME OF DOCTOR OF APPLICANT

to the corresponding fields:

- CITY OF EMP_ADDR OF EMPLOYEE
- STATE OF EMP_ADDR OF EMPLOYEE
- NAME OF DOCTOR OF EMPLOYEE

The value in `COMPANY of LAST_JOB of APPLICANT` is not copied anywhere, because `EMPLOYEE` does not directly include any view named `LAST_JOB`. Nothing is copied into `OFFICE of DEPARTMENT of EMPLOYEE` because `APPLICANT` does not directly include any view named `DEPARTMENT`.
`NAME of APP_NAME of APPLICANT` is not copied to `NAME of EMP_NAME of EMPLOYEE`, because the source and destination views do not *directly* include the `NAME` fields and the views that include them are named differently.

At this, fields not only should have the same name but in case when they are defined in the rule or in the procedure then the target and the source must contain the same field. In the example below error is generated because fields `i, j` of view `v_glob` are different from `i, j` of view `v_proc`.

Example: Mapping of fileds from different contexts is not allowed.

```
DCL
  i,j INTEGER;
  v_glob VIEW CONTAINS i,j;
ENDDCL

PROC proc_view : INTEGER
  DCL
    i,j INTEGER;
    v_proc VIEW CONTAINS i,j;
  ENDDCL

  MAP v_proc TO v_glob
    *> ERROR: 13305-S View V_PROC cannot be mapped to view V_GLOB.
       Same-typed fields view map is deprecated feature. <*

  PROC RETURN(0)
ENDPROC
```

Map Same-Typed Fields

This feature is deprecated since RFX CG0500.

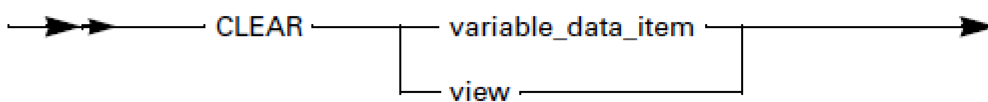
For example, a SMALLINT field is converted to an INTEGER and copied to an INTEGER field if both fields are the first (or second, or third...) fields in their respective views.

CLEAR Statement

A CLEAR statement sets the value of the specified variable to its initial value. Refer to Initializing Variables for information about initial values. If CLEAR is applied to a view, the result is the same as if CLEAR were applied to every field and subview of that view.

CLEAR Syntax

```
clear_statement:
      CLEAR variable_data_item
      CLEAR view
```



where:

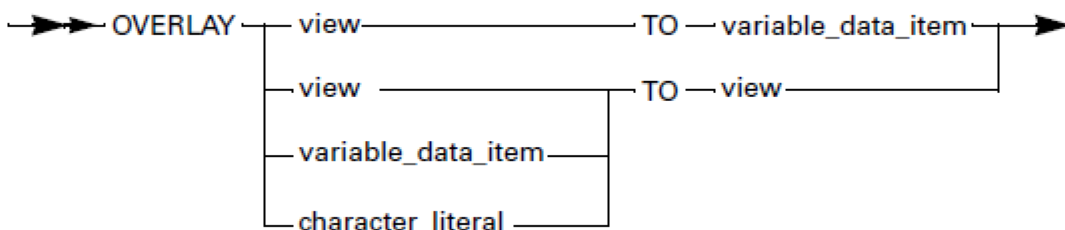
- variable_data_item – see Variable Data Item.
- view – see View.

OVERLAY Statement

An OVERLAY statement copies the value of the first item into the second item. The system determines the starting memory location of the first variable, copies that data into one block, determines the starting memory location for the second variable, and moves the block of data to that location. Thus, an OVERLAY action is a blind, byte-by-byte, memory copy. Although bypassing the safety mechanism of a MAP statement is one of the main purposes to use the OVERLAY statement, be aware of the platform-dependent storage differences. For example, because DEC data items are packed on the host, they have more bytes on the workstation than on a host.

OVERLAY Syntax

```
overlay_statement:
      OVERLAY view TO field
      OVERLAY source_for_view TO view

source_for_view:
      one of view expression field
```



where:

- variable_data_item – see Variable Data Item.
- view – see View.
- character_literal is a Character literal – see Character Value.

Valid combinations of OVERLAY statements are:

- `OVERLAY view TO view`

- `OVERLAY view TO field`
- `OVERLAY expression TO view`
- `OVERLAY field TO view`, where field is of type CHAR or VARCHAR.

Where *expression* can be any expression resulting in the value of type CHAR or VIEW except:

- HIGH_VALUE and LOW_VALUE
- aggregate

Regardless of the platform, the data items of the following types are allocated the same amount of memory, and it is safe to overlay one data item with another of the same type, *if* they have the same memory offset in the source and the destination:

- SMALLINT
- INTEGER
- DATE
- TIME
- TIMESTAMP
- CHAR

Special considerations exist when using the OVERLAY statement with the following data types:

- Using OVERLAY Statement with VARCHAR Data Types
- Using OVERLAY Statement with DEC and PIC Data Types
- Using OVERLAY Statement with Data Items of Different Length
- Using OVERLAY Statement in Multiple-Occurring Views

For language-specific considerations of the OVERLAY statements, refer to the following topics:

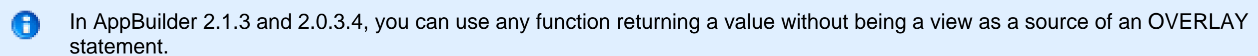
- OVERLAY Statements in C
- OVERLAY Statements in Java
- OVERLAY Statements in ClassicCOBOL
- OVERLAY Statements in OpenCOBOL

Example: Using OVERLAY with SMALLINT, INTEGER, and TIMESTAMP

If a rule contains the following declarations and initialization statements:

```
DCL
  V_int INTEGER;
  V_small_1, V_small_2 SMALLINT;
  V_stamp TIMESTAMP;
  V_view_1 VIEW CONTAINS V_int, V_stamp;
  V_view_2 VIEW CONTAINS V_small_1, V_small_2, V_stamp;
ENDDCL

MAP timestamp TO V_stamp OF V_view_1
```

It is safe to assume that the statement OVERLAY `V_view_1` TO `V_view_2` results in the correct value in `V_stamp` OF `V_view_2` and that this value is the same as the value of `V_stamp` OF `V_view_1` because both fields have the same memory offset 4 in the source and in the destination.



In AppBuilder 2.1.3 and 2.0.3.4, you can use any function returning a value without being a view as a source of an OVERLAY statement.

Example: SUBSTR () function as source of the OVERLAY statement for all platforms

```
OVERLAY SUBSTR(str_1, 1, 31) TO VIEW_1
```

Using OVERLAY Statement with VARCHAR Data Types

Use extreme caution when using an OVERLAY statement with VARCHAR data items. Internally, a VARCHAR length information is stored in its first two bytes. This is accounted for when you use a VARCHAR variable in a MAP statement, so this length information does not cause any problems. However, you must compensate for this fact when using an OVERLAY statement with a VARCHAR variable. Using an OVERLAY

statement with VARCHAR data is platform-dependent.

If the destination is a VARCHAR field, then an OVERLAY statement copies *n* bytes from the source (where *n* is the lesser of the two values: source length and maximum length of VARCHAR field) into the destination field, starting from the third byte. The first two bytes of the destination field contain value *n*.

Example: Using OVERLAY with VARCHAR Data Types

Example 1: OVERLAY Statement with VARCHAR data types
If a rule contains the declarations and initialization statements:

```
DCL
  V_varchar_5_1 VARCHAR(5);
  V_varchar_5_2 VARCHAR(5);
  V_varchar_9 VARCHAR(9);
  V_view_1 VIEW CONTAINS V_varchar_5_1;
  V_view_2 VIEW CONTAINS V_varchar_5_2;
  V_view_3 VIEW CONTAINS V_varchar_9;
ENDDCL

MAP 'Hello' TO V_varchar_5_1
```

then the statement

```
OVERLAY V_view_1 TO V_varchar_5_2
```

results in `V_varchar_5_2` having a length of 5 and containing '##Hel', where ## stands for two ASCII characters representing the ASCII value of the number of bytes copied from the source view `V_view_1`. The number of bytes cannot exceed the length of the destination, which is 5 in this instance. These ASCII characters would have binary values of 5 and 0.

Although tolerated by the Rules Language, putting the binary value 0 in a VARCHAR can cause some undesirable results. For example, the apparent truncation of data on display can affect system routines that treat the 0 as a string terminator.

The statement

```
OVERLAY V_view_1 TO V_varchar_9
```

results in `V_varchar_9` having a length of 7 and containing '##Hello', where ## stands for two ASCII characters representing the ASCII value of the length of `V_varchar_5_1` field. In this case, the ASCII characters would have binary values of 5 and 0.

However, the statements

```
OVERLAY V_view_1 TO V_view_2
OVERLAY V_view_1 TO V_view_3
```

result in `V_varchar_5_2` and `V_varchar_9` containing the same value 'Hello'. This happens because here the destination is seen as a *view* , rather than as a VARCHAR, and no offsetting is done to deal with the first two bytes of the VARCHAR.

Example 2: Comparing the results of MAP statement and OVERLAY statement

Suppose a rule contains the following declaration:

```
DCL
  CHAR_VAR_1 CHAR (5);
ENDDCL
```

and also includes the following view:

Then the statements:

```
MAP 'Hello' TO CHAR_VAR_1
MAP CHAR_VAR_1 TO VARCHAR_VAR_1
```

result in VARCH_VAR_1 containing 'Hello'. However, if you were to then use the following statement:

```
OVERLAY VIEW_C TO CHAR_VAR_1
```

the variable `CHAR_VAR_1` would contain '??HEL', where ?? stands for the ASCII value of the binary length of the VARCHAR variable. In this case, the ASCII equivalent of binary 5 would be null-ENQ.

Using OVERLAY Statement with DEC and PIC Data Types

Use extreme caution when using the OVERLAY statements with PIC and DEC data items. Only DEC or PIC data items that have the same length and scale should be used in the OVERLAY statements. No assumption can be made about the amount of memory allocated for the storage of DEC and PIC data items.

For example, although it might seem that DEC(27,20) and DEC(27,25) have the same representation because they have the same amount of memory allocated for their storage, the representation and contents are different; overlays between the two types result in invalid representation and corrupt data. AppBuilder is not designed to work with corrupted PIC and DEC data items.

For details about the Java generation of the OVERLAY statement with DEC and PIC data types, see also OVERLAY Statements in Java.

Using OVERLAY Statement with Data Items of Different Length

Use extreme caution when overlaying data items that have different lengths. If a data item used as destination in an OVERLAY statement is longer than the source data item, the semantics of the OVERLAY statement can vary on different platforms.

On some platforms, OVERLAY performs a byte-by-byte copy of the source then fills the rest of the target with blanks. On other platforms, however, the rest of the target remains unchanged. If the source is longer than the destination, then only the number of bytes equal to the length of the destinations is copied. This behavior should be taken into consideration when overlaying multiple-occurring views.

The OVERLAY statement for multiple-occurring views does not use the same algorithms as the MAP statement, and performs memory copy of all occurrences of the source view to the destination address.

For details about the Java generation of the OVERLAY statement with data items of different length, see also OVERLAY Statements in Java.

Example: OVERLAY Statement with data items of different length

The following example describes a situation where the OVERLAY statement fails. If a rule contains the following declarations and initialization statements:

```
DCL
  V_char_3 CHAR(3);
  V_char_2 CHAR(2);
  V_char_5 CHAR(5);
  V_smallint SMALLINT;
  V_view_1 VIEW CONTAINS V_char_3, V_char_2;
  V_view_2 VIEW CONTAINS V_char_5;
  V_view_11 VIEW CONTAINS V_view_1(10), V_smallint;
  V_view_22 VIEW CONTAINS V_view_2(9), V_smallint;
  I smallint;
ENDDCL

MAP 32 TO V_smallint OF V_view_22
DO TO 9 INDEX I
  MAP 'Hello' TO V_char_5 OF V_view_22 (I)
ENDDO
```

Then the statement

```
MAP V_view_22 TO V_view_11
```

results in V_smallint OF V_view_11 containing the value 32, and the rest of the fields in V_view_11 remain unchanged. However, the statement

```
OVERLAY V_view_22 TO V_view_11
```

results in:

- The first nine occurrences of `V_view_1` in `V_view_11` contain the value 'Hel' in the field `V_char_3` and the value 'lo' in the field `V_char_2` .
- The tenth occurrence `V_view_1` in `V_view_11` contains the value '##' in the field `V_char_3` (note that the third character is a blank symbol and '##' stands for two ASCII characters that represent binary value 32) and character value contains blanks in the field `V_char_2`.
- `V_smallint` contains value 8224 on the PC, which is an integer representation of the two bytes containing blanks.

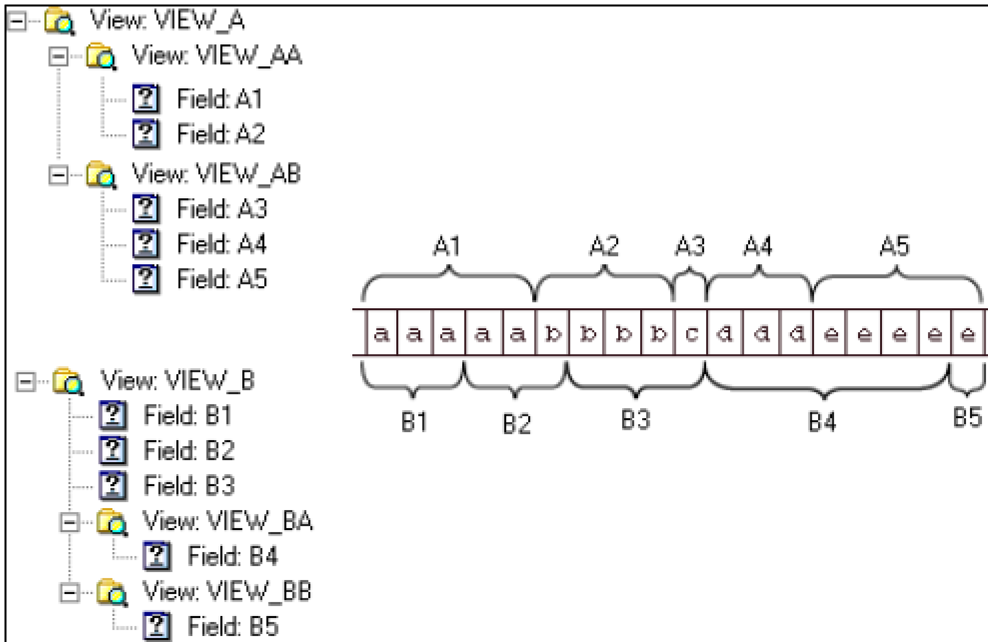
The value of `V_smallint` field after the overlaying `V_view_22` to `V_view_11` is 0 in Java generation.

Using OVERLAY Statement in Multiple-Occurring Views

The OVERLAY statement deals with multiple-occurring views in exactly the same way it does for non-multiple-occurring data items. In both cases, a block copy of memory occurs.

Results of OVERLAY VIEW_A TO VIEW_B shows the results of an OVERLAY from `VIEW_A` to `VIEW_B`.

Results of OVERLAY `VIEW_A` TO `VIEW_B`



The data in the fields of `VIEW_A` is copied and the copied data replaces the stored representation of `VIEW_B`. Where the structures of the two views differ, the data is divided differently into fields. For instance, the first ten characters of the stored representation of `VIEW_A` store the contents of the fields `A1`, `A2`, and `A3`: the strings 'aaaaa', 'bbbb', and 'c', respectively. However, when `VIEW_A` is overlaid on `VIEW_B`, those first ten characters in storage are used to fill fields `B1`, `B2`, and `B3`.

Because only the first three characters fit into `B1`, the last two characters of `A1` and the first character of `A2` are used for `B2`, and `B2` ends up holding 'aab'.

`B3` is defined to hold four characters: 'bbbc', which consist of the remaining three characters from `A2` and the single character of `A3`. OVERLAY is left justified, and characters that exceed the length of an overlaid field are truncated.

You can also OVERLAY a character expression to a view and a view to a character field. If `VIEW_A` in Results of OVERLAY VIEW_A TO VIEW_B was a character literal or a character field that contained the string 'aaaaabbbbcdddeeeee', `VIEW_B` would end up with the same data in the same fields. If `VIEW_A` was as shown, and `VIEW_B` was a character field at least 18 characters long, `VIEW_B` would be set to 'aaaaabbbbcdddeeeee'.

Redefining Views

You can have one view redefine another view, meaning that the data contained in the two views are stored at the same address in memory. Essentially, the two views are just different names for the same collection of data allowing you to use multiple definitions for the same memory space. This is an alternative to overlaying views, which copies the data from one area in memory to another thus creating two copies of the same data.

> Java does *not* support redefining views.

Procedure - Redefining a View with Another View

Follow these steps to have a view redefine another view:

Open the Construction Workbench and the Hierarchy window.

1. Create a "View Includes View" relationship between the two views, making the original view as the parent and the redefined view as the child.

   > Although these two views are created as parent and child, they are really clones – two names for the same view.

2. Double-click the relationship line between the two views to bring up the "Edit view includes" window.
3. Change the Null indicator property to Redefines View.

The following restrictions apply to using redefined views:

- The first view cannot be a locally-declared view.
- The length of the second view must be less than or equal to the length of the first view. This is not enforced, but you encounter errors if the second view exceed the length of the first view.

Condition Statements

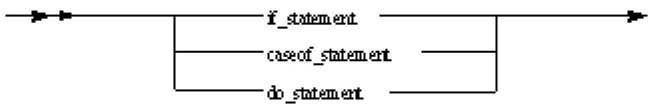
Condition statements direct processing control within a rule to one group of statements or another depending on the value of a condition. The following statements are described in this chapter:

- IF Statement
- CASEOF Statement
- DO Statement

Condition Statement Syntax

condition_statement:

one of if_statement caseof_statement do_statement



IF Statement

An IF statement routes control between two groups of statements, depending on the truth or falsity of a condition. If the condition is true, processing continues with the statements following the condition but before the ELSE clause (or before ENDIF, if there is no ELSE clause). If the condition is false, processing continues with any statements following the optional ELSE. If the condition is false and there is no ELSE, no statements are executed.
Upon completion, processing continues with the statement following ENDIF.
It is possible to nest IF statements. The nest depth depends on the target language (COBOL, C, or Java) compiler possibilities.

IF Statement Syntax

if_statement:

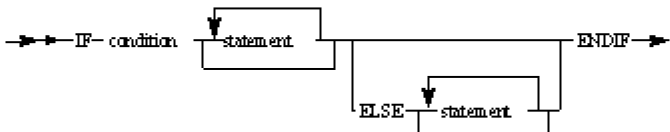
if_part [else_part] ENDIF

if_part:

IF condition statement_list

else_part:

ELSE statement_list



where:

- condition---see Condition Operators.
- statement is any Rules Language statement, except a declarative statement.

Examples: IF Statement and Nesting IF Statements

The following are examples of IF statement and nesting IF statements.

Example 1: IF Statement

```
IF EVENT_SOURCE OF HPS_EVENT_VIEW = 'UPDATE'
MAP CUSTOMER_DETAIL TO UPDATE_CUSTOMER_DETAIL_I
USE RULE UPDATE_CUSTOMER_DETAIL
ELSE
MAP 'NO_CHANGE' TO RETURN_CODE OF
DISPLAY_CUSTOMER_DETAIL_O
ENDIF
```

Example2 : Nesting IF Statements

```
IF EVENT_SOURCE OF HPS_EVENT_VIEW='UPDATE'
IF CUSTOMER_DETAIL <> UPDATE_CUSTOMER_DETAIL_I
MAP CUSTOMER_DETAIL TO UPDATE_CUSTOMER_DETAIL_I
USE RULE UPDATE_CUSTOMER_DETAIL
ELSE
MAP "No changes detected" TO UPDATE_STATUS OF CUSTOMER_WND_I
ENDIF
ELSE
MAP 'NO_CHANGE' TO RETURN_CODE OF DISPLAY_CUSTOMER_DETAIL_O
ENDIF
```

CASEOF Statement

A CASEOF statement routes control among any number of groups of statements, depending on the value of a field. The CASEOF statement is a shorter way of expressing the same flow of control that nested IF statements produce.
A CASEOF statement determines which one, if any, of the literals or symbols in its subordinate CASE clauses equals the value in the field in the CASEOF clause. Processing continues with the statements following that CASE clause. If none of the CASE clauses equal the value in the field, processing continues with the statements following the optional CASE OTHER. If none of the CASE clauses equal the value in the field and there is no CASE OTHER clause, no statements are executed.
Upon completion, processing continues with the statement following ENDCASE.

CASEOF Statement Syntax

caseof_statement:

CASEOF *field_name* (case_clause_part)* [caseother_part] ENDCASE

case_clause_part:

CASE selector (selector)* statement_list

caseother_part:

CASE OTHER statement_list

selector:

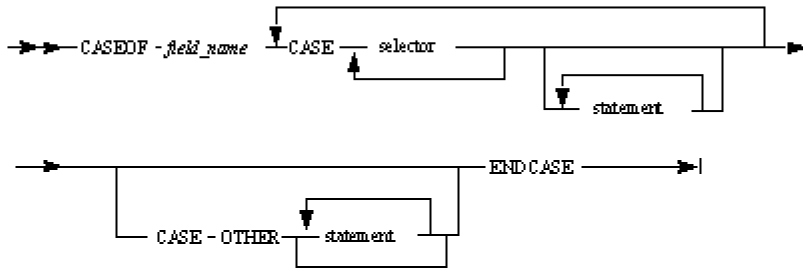
‘*character_literal*’

numeric_literal

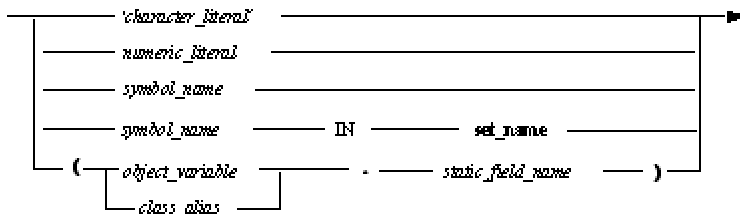
symbol_name [IN set_name]

‘(‘ *object_variable . static_field_name* ‘)’

‘(‘ *class_alias . static_field_name* ‘)’



where selector has the following form:



The last form of selector is available only in Java.
where:

- *symbol_name* – see Symbol.
- statement is a sequence of any Rules Language statement, except a declarative statement.
- *static_field_name* is the name of the field of a suitable type, that is, a type with constants that can appear as a selector. Allowable types are:
- Numeric
- Character

Usage

The literals or symbols in the CASE clauses must be compatible with the type of field in the CASEOF clause. In Java, ClassicCOBOL, OpenCOBOL, and C#, selector can be given by FUNC ('character_literal'), where FUNC is one of MIXED, DBCS or CHAR.
A literal or symbol can appear only once within the CASE clauses of a single CASEOF statement.
Thus, it is illegal to have:

```
CASE 5
statements
CASE 5
statements
```

in the same CASEOF statement. However, the CASE clauses need not contain or cover all possible values. In addition, as with any use of a string literal, a CASE clause is case-sensitive.

Also note that when a symbol has the same name as a rule, an ambiguity can arise in a CASE statement as to whether the symbol is meant or a rule. Such an ambiguity is always resolved in favor of set symbol. If rule call is intended, then USE RULE statement should be used. For example, code this:

```
CASE AMBIGUOUS_NAME
USE RULE AMBIGUOUS_NAME
```

rather than this:

```
CASE AMBIGUOUS_NAME
AMBIGUOUS_NAME
```

For specific considerations, refer to CASEOF in Java.

Example: CASEOF statements

In the statement

```
CASEOF LETTER
CASE 'A' 'E' 'I' 'O' 'U' 'a' 'e' 'i' 'o' 'u'
MAP 'Vowel' TO LETTER_TYPE
CASE 'Y' 'y'
MAP 'Sometimes vowel' TO LETTER_TYPE
CASE OTHER
MAP 'Consonant' TO LETTER_TYPE
ENDCASE
```

the character field LETTER_TYPE is set to one of the character literals 'Vowel', 'Sometimes vowel', or 'Consonant', depending on the value of the character field LETTER. The statement

```
CASEOF YEARS_AS_EMPLOYEE
CASE 5
MAP 'Certificate' TO BONUS_ITEM
CASE 10
MAP 'Plaque' TO BONUS_ITEM
CASE 25
MAP 'Watch' TO BONUS_ITEM
ENDCASE
```

sets BONUS_ITEM to the gift appropriate for an employee bonus after the indicated years of employment.

The following shows a skeleton example of the use of a CASEOF construct along with the semantically identical translation to a set of nested IF statements.

```
CASEOF TRANS_CODE
CASE 'A' 'C'
statement 1
statement 2
CASE 'M' 'U'
statement 3
CASE 'X'
statement 4
statement 5
CASE OTHER
statement 6
ENDCASE
```

The IF statement equivalent of the above is:

```
IF TRANS_CODE = 'A' OR TRANS_CODE = 'C'
statement 1
statement 2
ELSE
IF TRANS_CODE = 'M' OR TRANS_CODE = 'U'
statement 3
ELSE
IF TRANS_CODE = 'X'
statement 4
statement 5
ELSE
statement 6
ENDIF
ENDIF
ENDIF
```

DO Statement

A DO statement provides control for repetitive loops. If a DO statement contains a WHILE clause, any statements between DO and ENDDO are executed repetitively as long as the condition in the WHILE clause is true and the TO bound is not reached. When one of the conditions mentioned becomes false, control passes from the WHILE clause to the statement following the ENDDO.

DO Statement Syntax

do_statement:

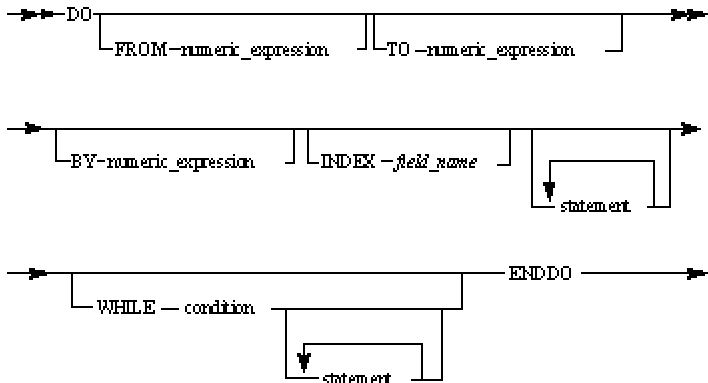
DO do_clauses [statement_list] [while_clause] ENDDO

do_clauses:

[FROM numeric_expression] [TO numeric_expression] [BY numeric_expression] [INDEX *field_name*]

while_clause:

WHILE condition [statement_list]



where:

- condition---see Condition Operators.
- numeric_expression---see Numeric Expressions.
- statement is any Rules Language statement, except a declarative statement.

Usage

A DO statement provides control for repetitive loops. If a DO statement contains a WHILE clause, any statements between DO and ENDDO are executed repetitively as long as the condition in the WHILE clause is true. When the condition becomes false, control passes from the WHILE clause to the statement following the ENDDO. The WHILE clause can be at the top of the loop, at the bottom of the loop, or anywhere in the middle. Nevertheless, if the WHILE condition is true but the ending value of a counter is reached (see TO clause), the loop finishes and control passes from the WHILE clause to the statement following the ENDDO.
A DO statement does not have to contain a WHILE clause. However, if it does not, it must contain at least one of the following four counter clauses that govern execution of the loop:

FROM clause

- Specifies the starting value for a counter.

TO clause

- Specifies the ending value of a counter.

BY clause

- Specifies how much to increment the counter with each execution of the loop.

INDEX clause

- Specifies the name of a field to use as the counter. Its value changes with each execution of the loop. FLOAT and DOUBLE fields are not allowed.

A DO statement with one of these clauses is called an indexed DO.

Indexed DO Statements

The following restrictions apply to an indexed DO statement:

- It must contain at least one FROM, TO, BY, or INDEX clause.
- If INDEX clause is present the expression following a FROM, TO, and BY must resolve to a counter type. If it does not, then the system converts the expression value to the counter type. If INDEX clause is not present then the type of the counter provided by system depends on the type of the expression following a FROM, TO, and BY.
- The system provides a default value for any FROM, BY, or TO clause if you do not provide one. The following table lists these default

values.

Statement	Default value
FROM	1
BY	1
TO	32,766

- The INDEX clause is optional. The system provides its own counter variable if you do not.
- The value in the TO clause is inclusive. Thus, a loop containing

DO FROM 1 TO 10

executes ten times.

If a BY clause forces the counter above the TO value, the loop is finished. Thus, a loop containing

DO FROM 1 TO 10 BY 4

executes three times (for the values 1, 5, and 9).

Upper bounds of FROM and BY clauses are defined by the type of the loop counter as follows: constants specified in these clauses cannot exceed the maximum value allowed for the type of the loop counter. For example, if SMALLINT counter is specified then constants in FROM and BY clauses cannot exceed value 32766. In other case an error is generated. Constants specified in the TO clause can exceed the maximum value for the type of the counter but in this case a warning is generated.

Platform specific information

For platform specific information, see Indexed DO Statements in Java.

DO Statement Restrictions

When generating the target code, the goal is to produce the most efficient and the most readable code, using the constructions of the target language as much as possible.
None of the target languages (COBOL, Java and C) supports detection of the overflow condition for the loop index. Because of this restriction, the Rules Language also does not support overflow detection.
For example, the execution of the following loops will never stop:

DO TO 32767 ENDDO
DO BY 3 TO 32765 ENDDO
DO BY 100 ENDDO

All of the above loops will have internal loop counter generated as 2-byte integer (SMALLINT type in Rules Language), which at some point will become a negative number because of the undetected overflow when adding BY value to the index.

This problem can only be seen in the following cases:

- when FROM, BY and TO values are such that after iteration number N:

BY > 0 and
FROM+BY*N <= TO and
FROM+BY*(N+1) > MAX

- or

BY < 0 and
FROM+BY*N >= TO and
FROM+BY*(N+1) < MIN

where:

- MAX is the maximum value for the data type used for the index: 32767 for smallint, 2147483647 for integer, 999 for DEC(3,0) and so on.
- MIN is the minimum value for the data type used for the index.

For platform specific restrictions, see DO Statements in OpenCOBOL.

Examples: Using DO Statements

For the first two examples, assume that *TOTAL_AMOUNT* = 2 and *TOTAL_LIMIT* = 1 before the loop executes.

Example 1
In Example 1, the preceding DO statement executes statements 1 and 2 once before leaving the loop.

```
DO
WHILE TOTAL_AMOUNT > TOTAL_LIMIT
statement 1
statement 2
MAP (TOTAL_AMOUNT - 1) TO TOTAL_AMOUNT
ENDDO
```

Example 2
In Example 2, the preceding DO statement executes statements 1 and 2 twice before leaving the loop.

```
DO
statement 1
statement 2
WHILE TOTAL_AMOUNT > TOTAL_LIMIT
MAP (TOTAL_AMOUNT - 1) TO TOTAL_AMOUNT
ENDDO
```

Example 3
In Example 3, the preceding DO statement executes from 1 to the value contained in LOOP_END by the value in STEP_VAR, incrementing COUNTER_VAR as it does so.

```
DO TO LOOP_END BY STEP_VAR INDEX COUNTER_VAR
statement 1
statement 2
ENDDO
```

Example 4
In Example 4, the FROM clause and statements 1 and 2 execute, and then the condition for the WHILE loop is checked. So long as the WHILE condition is true, the FROM loop controls processing. When the WHILE condition becomes false, control continues with the statement following the ENDDO.

```
DO FROM START_LEVEL INDEX COUNTER
statement 1
statement 2
WHILE CODES (COUNTER) <> TERM_CODE IN VALID_CODES
ENDDO
```

Transfer Statements

Transfer statements switch control of an application from one rule to another to perform another task, from a rule to a window to have the window appear on the screen, from a rule to a report to print the report, or from a rule to an internal procedure. Return statements return control to a rule. The following transfer statements are described in this chapter:

- USE Statements
- CONVERSE Statements
- RETURN Statement
- PERFORM Statement
- PROC RETURN Statement

Transfer Statement Syntax

transfer_statement:

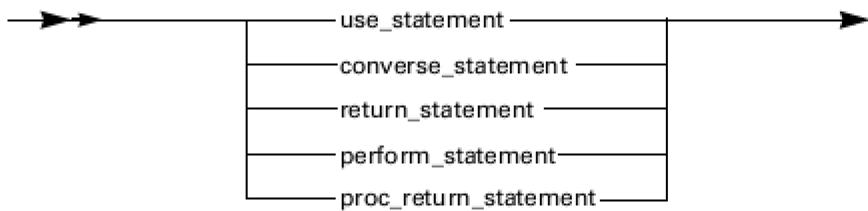
use_statement

converse_statementF

return_statement

perform_statement

proc_return_statement



USE Statements

A USE statement transfers the logic flow of an application to another rule or to a component. You can specify the input view directly in a RULE call. The called rule or component then directs control of the application. After it and any rules or components it calls finish processing, control returns to the calling rule. The calling rule resumes processing at the statement after the USE statement that invoked the called rule or component. The following topics are discussed in this section:

- USE RULE Statement
- USE RULE ... NEST Statement
- USE RULE ... DETACH Statement
- Passing Data to a Rule
- USE RULE ... INIT Statement
- USE COMPONENT Statement

USE Statement Syntax

use_statement:

[USE RULE] *rule_name* ['(' rule_comp_parameter ')'] [nest_detach_init_part]

nest_detach_init_part:

NEST [INSTANCE file_name]

DETACH [INSTANCE file_name] [OBJECT field_name]

INIT [TERMINAL character_expression] [start_clause numeric_value]

start_clause:

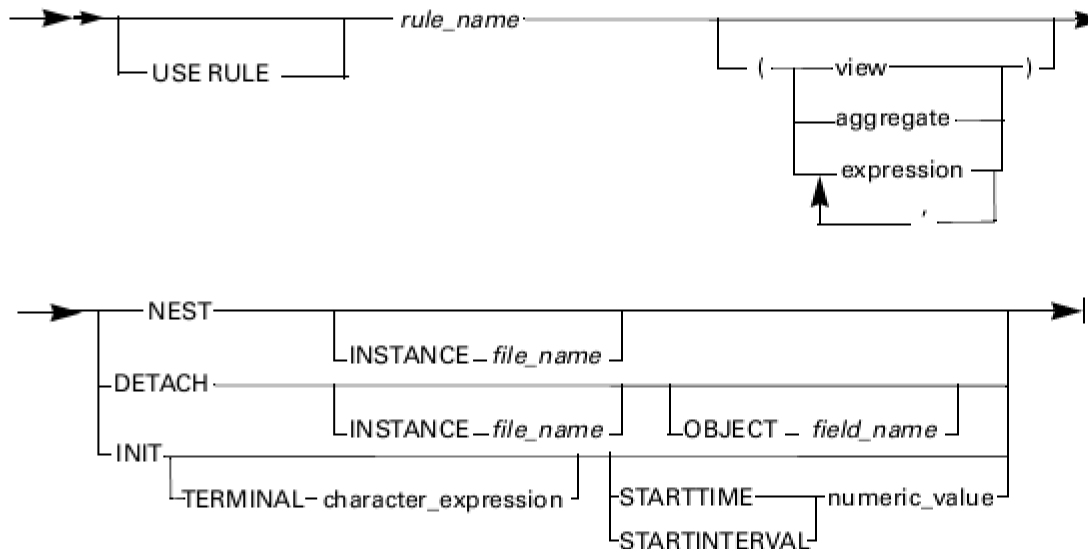
one of STARTTIME STARTINTERVAL

rule_comp_parameter:

view

aggregate

expression (, expression)*



where:

- character_expression---see Character Expressions.
- numeric_expression---see Numeric Expressions.
- numeric_value---see Numeric Value.
- view---see View.

USE RULE Statement

A USE RULE statement transfers control, or *calls* , to another rule. There are several variations of the USE RULE statement:

- A simple USE RULE statement invokes another rule without any special instructions. If the called rule converses a window, all other windows in its application are removed before its window appears. Its window is modal; that is, users cannot return to the previous window until they perform some action in that window.
- A USE RULE..NEST statement invokes another rule, and if the called rule converses a window, it instructs the called rule to overlay its window over other windows of the application that are currently visible. That is, the windows displayed to the user are "nested" one on top of the other. A nested window is also modal.
- A USE RULE..DETACH statement invokes another rule and instructs the called rule to share control with the calling rule. Any window the called rule converses is still nested, but is non-modal. Thus, a user can switch between that window and any other currently visible window at any time by simply selecting something in the desired window. An INSTANCE clause after a USE RULE?DETACH statement allows "multiple occurrences" of the same window to be displayed at the same time. A window conversed by a detached rule is also called a non-modal secondary window. Refer to "Event-driven Processing" in the *Developing Applications Guide* for more information.
- A USE RULE..INIT statement spawns an independently running AppBuilder host online rule.

The using clause is optional, depending on the existence of an input or an output view.
The called rule defines the input and output views in its linkage section and creates the linkage using the procedure division statement.

Invoking Subrules

Rules Language provides two methods to invoke subrules:

- USE RULE statement
- a "procedure call"-like expression

If a field, view or procedure hides the rule name, it can still normally be used in a USE RULE statement.

Notes for USE RULE

When a rule is invoked, data item initialization is performed. For details about data item initialization, refer to Initializing Variables.
See USE RULE Statement in OpenCOBOL for OpenCOBOL specific information.

For information about using display rules for the thin client, refer to the *Developing Applications Guide* .

A rule can always use another rule if they both have the same execution environment. However, a rule generally cannot use a rule with a different execution environment, except as Rule Using Rule Support shows.

Execution Environments shows the abbreviations used in Rule Using Rule Support.

Rule Using Rule Support shows the possible combinations of Rule using Rule when the execution environment of each rule is specified as a property of the Rule. You can also specify the execution environment of a rule by using a Partition. If you attach a Rule to a Partition (through its parent process), the rule's execution environment is that of the Machine entity that is associated with the Partition. When you prepare a Rule in configuration mode (using Partitions), the execution environment specified for the Partition overrides the execution environment specified for any rule attached to the Partition.

You can prepare client-side rules of a distributed application without using Partitions only if the ALWAYS_USE_DNA key in the [AE runtime] section of the client side Hps.ini file is set to YES at runtime. Because this setting causes a significant performance degradation, we recommend using it only in a development environment while testing small portions of your application.

Execution Environments

Abbreviation	Execution environment	Explanation
PC	PC Workstation	Workstation online rule
PCCICS on PC	PC & IBM Mainframe (CICS)	When prepared on a workstation, this kind of rule is treated like a PC Workstation rule
PCCICS on MVS	PC & IBM Mainframe (CICS)	When prepared on the host, this kind of rule is treated like an IBM Mainframe (CICS) rule
CICS	IBM Mainframe (CICS)	MVS host online rule
CICS&Batch	IBM Mainframe (CICS & Batch)	Either MVS batch or CICS, depending upon the environment of the calling rule
Batch	IBM Mainframe Batch (MVS)	Batch mode under MVS
IMS	IBM Mainframe (IMS)	MVS host online rule
PCIMS on PC	PC & IBM Mainframe (IMS)	When prepared on a workstation, this kind of rule is treated like a PC Workstation rule
PCIMS on MVS	PC & IBM Mainframe (IMS)	When prepared on the host, this kind of rule is treated like an IBM Mainframe (IMS) rule

Rule Using Rule Support

	Called Rule								
Calling rule	***PC***	***PCCICS on PC***	***PCCICS on MVS***	***CICS***	***CICS&Batch***	***Batch***	***IMS***	***PCIMS on PC***	***PCIMS on MVS***
PC	Y	Y	Y	Y	Y[a]	N	Y	Y	Y
PCCICS on PC	Y[b]	Y	Y	Y	C	N	Y	Y	Y
PCCICS on MVS	N	N	Y	Y	C	N	N	N	N
CICS	N	N	Y[b]	Y	C	N	N	N	N
CICS&Batch	N	N	N	W[c]	Y[a]	W[c]	N	N	N
Batch	N	N	N	N	B	Y	N	N	N
IMS	N	N	N	N	N	N	Y	N	Y[b]
PCIMS on PC	Y	Y	Y	N	N	N	Y	Y	Y
PCIMS on MVS	N	N	Y	N	N	N	Y[d]	N	Y
Y=Yes (valid combination) C=CICS B=Batch N=No (invalid combination) W=Warning (valid but be careful)									

a) *The way a called rule with an execution environment of CICS&Batch is executed depends on its calling rule. If the calling rule is online (PC, PCCICS, or CICS), the called rule is executed as a CICS program. If the calling rule is Batch, the called rule is executed as an MVS batch*

program. If the calling rule is itself CICS&Batch, then the execution mode of the called rule is determined by going up the application hierarchy until a calling rule is found that is either online or Batch.
b)*This combination runs but is not recommended because it is not portable. If the calling rule is prepared instead on the MVS host, then it cannot call the PC rule because a host rule cannot call a workstation rule.*
c)*A warning is issued if a CICS&Batch rule calls either a CICS rule or a Batch rule. This is because a CICS&Batch rule inherits its execution mode from its calling rule (see note 1); therefore, the following combinations can result but would not run: An online rule calls a PCCICS rule, which calls a Batch rule. A Batch rule calls a PCCICS rule, which calls a CICS rule.*
d)*This combination runs but is not recommended because it is not portable. If the calling rule is prepared instead on the MVS host, then it cannot call the PC rule because a host rule cannot call a workstation rule.*

USE RULE ... NEST Statement

Use NEST only with a rule that converses a window. There is no imposed limit on the number of windows that can be nested on a workstation, although memory determines the practical limit. Typically, 15 nested windows is a practical limit. In 3270 Converse applications, you can nest only one window.

USE RULE ... DETACH Statement

Use DETACH only with a rule that converses a window. Both the calling rule and the called rule must be PC Workstation rules.

An INSTANCE clause creates a unique instance of the rule. The character value in the INSTANCE clause is the instance name and must be a set symbol, a literal, a MIXED field, or a character field up to 30 characters long.

The following restrictions apply to detached rules:

- There can be only five levels of detached rules.
- Detached rules cannot use Dynamic Data Exchange (DDE).
- 3270 Converse applications do not support modeless windows.

USE RULE ... DETACH and USE RULE ... NEST statements for OpenCOBOL generate a normal rule call, also issuing a WARNING message that informs you what had been done.

See Use RULE ... DETACH OBJECT statement in Java for Java specific information.

Passing Data to a Rule

AppBuilder Rules Language supports two methods of passing data to a rule invoked with a USE RULE statement:

- Mapping Data to the Input View
- Passing Data in the USE RULE Statement

Mapping Data to the Input View

One method of passing data to a rule is to map the data into the input view of the called rule in a previous assignment statement, as shown in the following example:

```
MAP CUSTOMER_DETAIL TO UPDATE_CUSTOMER_DETAIL_I
USE RULE UPDATE_CUSTOMER_DETAIL
 IF RETURN_CODE1 OF UPDATE_CUSTOMER_DETAIL_O <> 'FAILURE'
 MAP 'UPDATE' TO RETURN_CODE OF
 DISPLAY_CUSTOMER_DETAIL_O
ENDIF
```

If the HPS_EVENT_VIEW registers that the *Update* menu choice is selected, the rule calls UPDATE_CUSTOMER_DETAIL, which stores the data from CUSTOMER_DETAIL to a file. However, before it invokes UPDATE_CUSTOMER_DETAIL, the rule maps the data from the window view CUSTOMER_DETAIL into UPDATE_CUSTOMER_DETAIL_I, the input view of the rule UPDATE_CUSTOMER_DETAIL.

DISPLAY_CUSTOMER_DETAIL maps either 'UPDATE' or 'NO_CHANGE' into its own output view to tell the rule that called it whether the data from the window have been stored.

Passing Data in the USE RULE Statement

A second method of passing data to a rule is to include the data in the USE RULE statement itself. For example, suppose the rule INTEGER_SUM has an input/output view named INTEGER_SUM_IO_VIEW containing the integers p1, p2, and res. Another rule can invoke INTEGER_SUM as follows:

```
DCL
I1, I2 INTEGER;
V VIEW CONTAINS P1, P2, RES;
ENDDCL
...
*>OLD VARIANT<*
MAP I1 TO P1 OF INTEGER_SUM_IO_VIEW
MAP I2 TO P2 OF INTEGER_SUM_IO_VIEW
USE RULE INTEGER_SUM


*>NEW VARIANT 1<*
USE RULE INTEGER_SUM (I1, I2, 0)


*>NEW VARIANT 2<*
MAP I1 TO P1 OF V
MAP I2 TO P2 OF V
USE RULE INTEGER_SUM (V)


*>NEW VARIANT 3<*
USE RULE INTEGER_SUM (2*I1, 34*I2, 0)
```

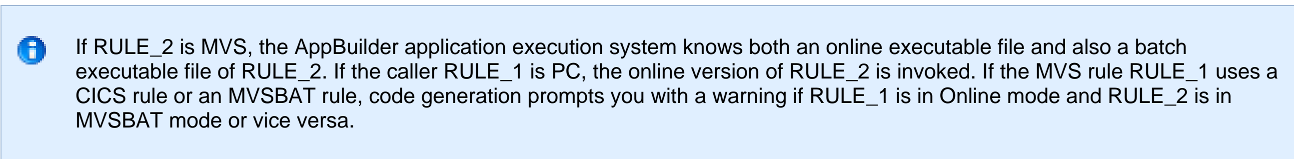
Restrictions on Use

Whether RULE_1 can use RULE_2 depends on the execution environments of both the "caller," RULE_1, and the "called," RULE_2. Rules Using Rules shows the valid interrelationships between calling and called rules---in terms of the execution environment. The choices of execution environments include the following:

- PC (also referred to as "Workstation")
- CICS (host online)
- MVSBAT (pure batch mode under MVS, executed through JCL)
- MVS (rules/components that can be used either in MVSBAT or CICS mode)
- IMS PC, CICS, and IMS are online environments. MVSBAT is a batch environment. MVS means either CICS or MVSBAT, depending on the nature of the caller. If the caller RULE_1 is online, the MVS rule or component RULE_2 is to be executed as a CICS program. If RULE_1 is MVSBAT, RULE_2 is to be executed as an MVSBAT program. If RULE_1 is MVS, the online or batch nature of RULE_2, and also RULE_1, is inherited from the caller of RULE_1.

Rules Using Rules

Calling Rule Type	**Called Rule type**				
	PC	**CICS**	**MVSBAT**	**MVS**	**IMS**
PC	Y	Y	N	Y	Y
CICS	N	N	N	O	N
MVSBAT	N	N	Y	B	N
MVS	N	W	W	Y	N
IMS	N	N	N	N	Y
Y=Yes (valid combination) N =(invalid combination) O=Online B=Batch W=Warning (valid but use caution)					


If RULE_2 is MVS, the AppBuilder application execution system knows both an online executable file and also a batch executable file of RULE_2. If the caller RULE_1 is PC, the online version of RULE_2 is invoked. If the MVS rule RULE_1 uses a CICS rule or an MVSBAT rule, code generation prompts you with a warning if RULE_1 is in Online mode and RULE_2 is in MVSBAT mode or vice versa.

For purposes of Rules Using Rules, a rule that has an environment of PCCICS is the same as a PC rule if prepared on a workstation and is the same as a CICS rule if prepared on the host. Likewise, a PCIMS rule is the same as a PC rule if prepared on a workstation and an IMS rule if prepared on the host.

USE RULE ... INIT Statement

A USE RULE...INIT statement initiates the execution of the called rule, and any rules and components it calls, and causes the called rule to run independently from the calling rule. The initiated rule must be a host online rule (with which it has an existing relationship), with one of the IBM mainframe execution environments, either CICS, CICS & Batch, or IMS. You can use a USE RULE...INIT statement by itself or with the following clauses:

- A TERMINAL clause specifies the terminal on which the initiated rule is to run. The character value within the clause is the ID of the terminal; only the first four characters are recognized. You can initiate a rule on only one terminal with each USE RULE..INIT TERMINAL statement, and that terminal must be signed on at the time the statement is executed.
- A STARTTIME clause indicates a specific time for the execution of the initiated rule. The numeric value within the clause indicates the time when the rule starts to execute.
- A STARTINTERVAL clause delays the execution of the initiated rule. The numeric value within the clause indicates how long from the execution of the statement until the rule starts to execute.

The numeric values in the STARTTIME and STARTINTERVAL clauses are the concatenation of three non-negative integers (*hh, mm, ss*) such that:

- *hh* is between 0 and 23 (hours)
- *mm* is between 0 and 59 (minutes)
- *ss* is between 0 and 59 (seconds)

Notes for USE RULE...INIT

The AppBuilder code generator cannot validate TERMINAL, STARTTIME, and STARTINTERVAL clauses because they can be assigned dynamically.

For example:

```
MAP 'Nonsense Terminal ID' TO TERMINAL_ID
MAP -25616199 TO START_TIME
USE RULE RULE200 INIT
TERMINAL TERMINAL_ID
STARTTIME START_TIME
```

The preceding rule would prepare cleanly but might encounter problems at runtime.

Other conditions and restrictions for using a USE RULE...INIT statement depend on the execution environment of the initiating rule, either CICS, CICS and Batch, or IMS.

CICS and Batch Execution Environment

A mainframe rule can initiate only a CICS rule or a CICS and Batch rule.

CICS Execution Environment

A CICS rule can initiate only a CICS rule or a CICS and Batch rule.
The TERMINAL clause cannot be combined with either the STARTTIME or STARTINTERVAL clauses.

IMS Execution Environment

In IMS, rules have a processing type in addition to an execution environment. IMS Rule Processing Types summarizes whether a rule of one processing type can use a USE RULE..INIT statement to call a rule of another processing type.
A DL/I Batch rule in an IMS environment initiated by a USE RULE...INIT statement cannot contain another USE RULE..INIT statement within it.

Do not use a USE RULE..INIT statement for multiple calls to a batch rule in IMS, because the IMS Run Control program starts a batch job member for each call. Use a USE RULE statement instead.

The TERMINAL, STARTTIME, and STARTINTERVAL clauses are not supported for rules operating under IMS.

IMS Rule Processing Types

	Called Rule Processing Type			
Calling Rule Processing Type	**MPP**	**Conversational**	**BMP**	**DL/I Batch**
MPP	Y	N	Y	Y
Conversational	Y	Y	Y	Y
BMP	Y	N	Y	Y
DL/I Batch	N	N	N	N
Y=Yes (can call) N=No (cannot call)				

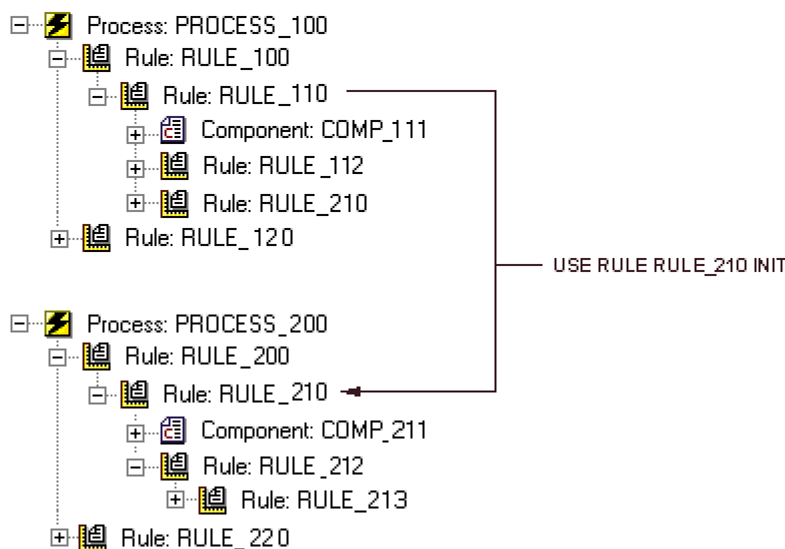
Example: USE RULE...INIT

Given an application consisting of the entities and relationships shown in the figure below, the statement (coded within rule RULE_110):

USE RULE RULE_210 INIT

initiates the execution of RULE_210. However, processing does not start with the root PROC_200, but rather with RULE_210. RULE_210 uses COMP_211 and RULE_212, which, in turn, can use other rules and components. Meanwhile, execution of RULE_110 proceeds independently: RULE_110 uses COMP_111, then RULE_112, then returns control to RULE_100, and so on. RULE_210 does not return control to RULE_110. RULE_110 continues to operate independently.

Sample Hierarchy for USE RULE...INIT



To have RULE_210 start after 4 hours and 8 seconds, change the USE RULE statement to read:

USE RULE RULE_210 INIT STARTINTERVAL *040008*

Alternatively, to have RULE_210 start at 3:00:41 PM, change the USE RULE statement to read:

USE RULE RULE_210 INIT STARTTIME *150041*

To link the forked-off processing of RULE_210 to a specific workstation ID, for example, 'wwww', change the USE RULE statement to read:

USE RULE RULE_200 INIT TERMINAL 'wwww'

USE COMPONENT Statement

A USE COMPONENT statement passes processing control to a component. A component is a programming module that is coded in some

language other than the Rules Language, such as C or COBOL. Refer to *Developing Applications Guide* for more information about writing components. Refer to *System Components Reference Guide* for more information about using components provided with AppBuilder.

Mainframe notes
Preparing a C language component that has a host execution environment creates only an MVSBATCH executable. * 3270 converse mainframe system components are not supported for OpenCOBOL.

A rule can always use a component if they both have the same execution environment. However, a rule generally cannot call a component with a different execution environment except as shown in Rules Component Support. The execution environments on the table include:

- PC (also referred to as "Workstation")
- CICS (host online)
- MVSBAT (pure mainframe batch mode)
- MVS (component that can be used either in MVSBAT or CICS mode)
- IMS
- Java (PC execution environment and Java language, while PC refers to other language available on workstation (C))
- PCCICS (When prepared on a workstation, this kind of rule is treated like a PC Workstation rule. When prepared on the host, this kind of rule is treated like an IBM Mainframe (CICS) rule.)

Rules Component Support

	Components					
Rules	**PC**	**CICS**	**MVSBAT**	**MVS**	**IMS**	**Java**
PC	Y	N	N	Y	N	N
CICS	N	Y	N	C	N	N
MVSBAT	N	N	Y	B	N	N
MVS	N	W	W	Y	N	N
IMS	N	N	N	N	Y	N
Java	Y	N	N	N	N	Y
Y=Yes (valid combination) N=No (invalid combination) C=CICS B=Batch W=Warning (valid but use caution)						

For information about calling user components in ClassicCOBOL and OpenCOBOL, refer to User Components in ClassicCOBOL and OpenCOBOL.

CONVERSE Statements

The following CONVERSE statements are discussed in this section:

- CONVERSE WINDOW Statement
- CONVERSE REPORT Statement
- CONVERSE for Global Eventing

CONVERSE Statement Syntax

converse_statement:

CONVERSE WINDOW *window_name* [NOWAIT]

CONVERSE REPORT *report_name* conv_report_subclause

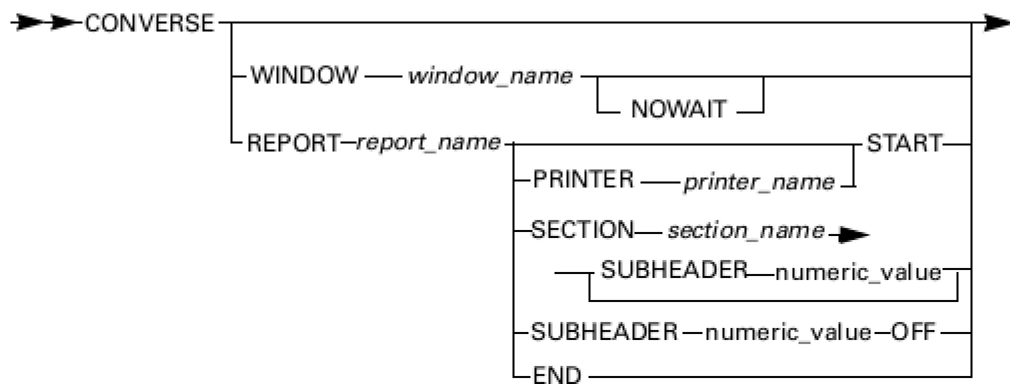
conv_report_subclause:

[PRINTER *printer_name*] START

SECTION *section_name*

SUBHEADER numeric_value [END]

END



where:

- numeric_value---see Numeric Value.
- *printer_name* is a character value containing the printer name---see Character Value.
- *report_name* is the name of the report, which belongs to the current rule.
- *section_name* is the name of the section, which belongs to the report (report_name).
- START is optional if the report has no sections attached.

A CONVERSE statement performs one of the following actions:

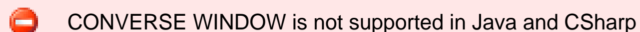
- Displays a window.
- Prints a report or portion of a report.
- Blocks a rule until it receives an event.

Upon completion of these actions, control automatically returns to the rule containing the CONVERSE statement.

CONVERSE, CONVERSE WINDOW, and CONVERSE REPORT are not supported in OpenCOBOL.

CONVERSE WINDOW Statement

The CONVERSE WINDOW statement causes the named window entity's panel to display on the screen, so that a user can manipulate the window's interface and field data. In the rules code, execution remains on the CONVERSE WINDOW statement until an event is returned to the rule. In other words, a CONVERSE WINDOW statement "waits" for an event and then continues with the execution of the rule. Manipulating a control object on the window interface generates a user interface event. This returns control to the rule. A system event or an event from a parent or child rule also causes a rule to leave its waiting state. Refer to the *Developing Applications Guide* for a detailed explanation of events and event-driven processing.


CONVERSE WINDOW is not supported in Java and CSharp

NOWAIT

A CONVERSE WINDOW...NOWAIT statement causes the AppBuilder environment to converse a window and return control immediately to the rule containing the statement. In other words, it does not wait for an event before continuing to process the rule.

Notes for CONVERSE WINDOW

A given rule can have a converse relationship with only one window. Thus, while a rule can have more than one CONVERSE WINDOW statement, each statement must refer to the same window.

Procedure - CONVERSE WINDOW

Use a CONVERSE WINDOW statement in the following series of actions:

1. Map the data in the fields in the window to the window's input view.
2. Converse the window.
3. Examine the fields in the rule's HPS_EVENT_VIEW view to decide what further action to take.

CONVERSE REPORT Statement

A rule can control the printing of a report by conversing a report entity. It is possible to converse on the mainframe in ClassicCOBOL (Batch and CICS) and on a workstation in Java.

This section applies to ClassicCOBOL and Java only. It is *not* supported in C, OpenCOBOL, or CSharp.

When printing reports on the host or on the workstation in Java, each report entity contains one or more section entities, each of which includes layout data for part of the report. A rule can map data into the view of a section and converse the report that contains that section to print the section's data. You can print a whole report by issuing a series of CONVERSE REPORT statements. Conversing a single section might produce as little as a single line of a report.

START and PRINTER

A CONVERSE REPORT...START statement initiates the printing of a report. A rule must contain one of these statements before any other CONVERSE REPORT statements. A CONVERSE REPORT...START statement sets up global storage for the named report, which is needed in all of the other CONVERSE REPORT statements. You can get a system abend without this statement. In order to print in CICS, include the PRINTER keyword and specify a four-character printer name after it. This is not necessary to print a batch or Java report.

For additional information refer to CONVERSE REPORT Statement in Java.

SECTION

Adding this keyword to a CONVERSE REPORT statement prints the data of the named section.

END

A CONVERSE REPORT?END statement prints the final break sections and terminates the printing of a report. A rule should contain one of these statements after all other CONVERSE REPORT statements.

On the host, a CONVERSE REPORT?END statement releases the global storage that was allocated in the corresponding CONVERSE REPORT?START statement. Lack of this statement does not cause an abend but you are not alerted to the fact that the final break section is missing.
It is a good practice to finish printing the report with CONVERSE REPORT...END statement on host and Java on the workstation. On the workstation in Java, if you do not issue a CONVERSE REPORT...END in the rule, the last page of the report won't be printed. This might lead to some undesirable side effects if you use the same report in this application again.

SUBHEADER

A CONVERSE REPORT...SECTION...SUBHEADER statement dynamically alters the subheader level number of the named section at execution time to the value in the SUBHEADER clause. The section retains this new number until another such statement redefines the number, the report ends, or a CONVERSE REPORT...SUBHEADER OFF statement is executed.

SUBHEADER OFF

A CONVERSE REPORT...SUBHEADER OFF statement dynamically resets to zero any subheader sequence number that is at least as big as the value following SUBHEADER. In other words, any regular section with a subheader sequence number greater than or equal to this value loses the subheader property.

Notes for CONVERSE REPORT

A rule executing a CONVERSE REPORT statement temporarily relinquishes control to the report. On the host and in Java on the workstation, when the report finishes processing the statement, it returns information to the calling rule in a special view called the Report Communication Area (RPTCA). You can view the RPTCA in the enterprise repository as you would any other view. Do not change the RPTCA (assuming you have the authorization), because AppBuilder references its members for internal processing.
If you have distributed the printing of a report over more than one rule in your application, you must issue a CONVERSE REPORT...START statement only once, not once per rule. Issuing a subsequent CONVERSE REPORT...START statement restarts a report and resets the page and line numbers to one. Similarly, if you have distributed printing over more than one rule, you need to issue a CONVERSE REPORT...END statement only once, not once per rule.

Example: CONVERSE REPORT

Assume you have a report called CUST_SALES_TRANS, with sections called RETAIL_CUST and WHOLESALE_CUST, and that you want to print it on a host printer called MAIN_PRINTER_4.
If this is a batch report, use the following statement to start printing the report:

CONVERSE REPORT CUST_SALES_TRANS START

If this is a CICS report, use the following statement instead:

CONVERSE REPORT CUST_SALES_TRANS PRINTER MAIN_PRINTER_4 START

After you have started printing, you can insert other CONVERSE REPORT statements as needed for your report.

CONVERSE REPORT CUST_SALES_TRANS SECTION RETAIL_CUST SUBHEADER 2

This statement resets the RETAIL_CUST section subheader level number to 2.

CONVERSE REPORT CUST_SALES_TRANS SECTION RETAIL_CUST
CONVERSE REPORT CUST_SALES_TRANS SECTION WHOLESALE_CUST

These statements print both sections of the report.

CONVERSE REPORT CUST_SALES_TRANS SUBHEADER 2 OFF

This statement resets the subheader level of the RETAIL_CUST section (and any higher-numbered sections) to zero.

CONVERSE REPORT CUST_SALES_TRANS END

This statement prints the final break sections and ends printing.

CONVERSE for Global Eventing

Global eventing provides a mechanism for passing messages among rules on the same or different systems. When one rule that includes an event in its hierarchy posts a message, any client rule that includes the same event in its hierarchy receives and processes the message. Execution of a CONVERSE statement without a window or report has the effect of blocking a rule until an event is received. When an event is received, the rule begins executing the statements following the CONVERSE.

You can also use a CONVERSE WINDOW statement to receive global events. A rule containing a CONVERSE WINDOW statement is unblocked upon receipt of global events as well as interface and system events---the statements following the CONVERSE begin executing when a global event is received.

Refer to the *Developing Applications Guide* for an explanation of global eventing.

Global eventing is *not* supported in OpenCOBOL generation.

RETURN Statement

A RETURN statement sends processing control back from the rule in which it appears to the rule that called its rule. If a called rule has no RETURN statement, processing control returns to its calling rule only after the last statement in the called rule is executed. Use a RETURN statement to send the control back to the calling rule before all lines in the called rule have been executed.

A RETURN statement inside a procedure causes a return from the rule containing the procedure. Use a PROC RETURN statement to return from the procedure to the point of invocation within the rule.

RETURN Statement Syntax

return_statement:

RETURN

```
►►─────────── RETURN ───────────────►◄
```

Example: RETURN Statements

In the following sample code, a portion of rule RULE_1 calls the rule RULE_2. The code in rule RULE_2, depending on ACTION_TO_PERFORM field of its input view, either performs a local procedure TEST and sets "return code" (STATUS field of RULE_2 output view) to 'TESTED' or

returns processing control to RULE_1, if 'SKIP' operation is chosen by RULE_1.

Rule 1

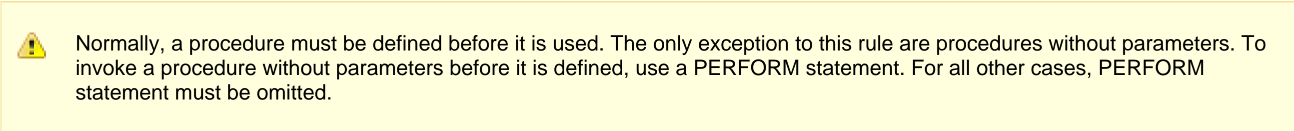
```
USE RULE RULE_2
MAP RULE_2O TO RULE_3I
```

Rule 2

```
CASEOF ACTION_TO_PERFORM OF RULE_2I
CASE 'TEST'
TEST(RULE_2I)
MAP 'TESTED' TO STATUS OF RULE_2O
RETURN
CASE 'SKIP'
RETURN
ENDCASE
```

PERFORM Statement

A PERFORM statement invokes a procedure within the same rule. When the statements of the procedure finish executing, control is returned to the statement following the perform statement. PERFORM statements allow you to invoke a procedure multiple times within a rule, rather than duplicating the statements of the procedure at multiple places in the rule.



Normally, a procedure must be defined before it is used. The only exception to this rule are procedures without parameters. To invoke a procedure without parameters before it is defined, use a PERFORM statement. For all other cases, PERFORM statement must be omitted.

PERFORM Statement Syntax

perform_statement:

[PERFORM] *procedure_name* [parameter_list]

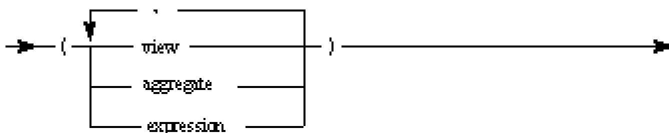
parameter_list:

'(' parameter (, parameter)* ')'

parameter:

one of view aggregate expression

where parameter_list can be:



where:

- expression---see Expressions and Conditions.
- view---see View.

For additional information see PERFORM Statement (PROC) in OpenCOBOL.

PERFORM Usage

Individual data items, views, or literals can be passed as parameters to a procedure. When a view is passed to a procedure, the view must be declared inside the procedure receiving it (see Common Procedure).
Parameters, including views, are passed by value only. That is, a variable cannot be passed to a procedure and expect it to be modified when the procedure returns. Parameters are input only. Do not pass an input/output parameter to a procedure.

If an object (see Object Data Types) is passed as a parameter to a procedure, although the pointer itself is passed by value, it still provides addressability to the object to which it points. Therefore, objects used as parameters allow a procedure to modify objects other than those that are passed in as parameters.

A procedure can return a value if that value is declared inside the procedure. If the procedure returns a value, the procedure is treated like a function and can be used in any context in which a function can be used.

Recursion is not supported for procedure calls in ClassicCOBOL or OpenCOBOL. There will be no error message if recursion is used, but execution results will be unpredictable.

Examples: PERFORM Statement: Error Code Processing and Using Procedures as Functions

Example 1: Simplifying error code processing

The following example illustrates the use of a procedure to simplify multiple error code processing; the coding of each process is simplified.

```
PROC handleError(errorCode SMALLINT)
DCL
  errorDescr VARCHAR(255);
ENDDCL

IF errorCode <= 0
  MAP "SUCCESS" TO errorDescr
ELSE IF errorCode <= 2
 MAP "WARNING" TO errorDescr
ELSE
 MAP "SEVERE ERROR" TO errorDescr
 ENDIF
ENDIF

PRINT errorDescr
ENDPROC

handleError(code)
handleError(dbCode)
```

Another way to simplify rule source is to use a macro for frequently repeated code. For information, see Substituting Rule Source Code.

Example 2: Using a procedure as a function
The following example illustrates using a procedure as a function. The procedure "cubed" receives one parameter (an integer) and returns the cube of that number. The procedure can be used in any context in which a function can be used---in this case in a MAP statement.

```
PROC cubed (inputNumber INTEGER): INTEGER
 PROC RETURN (inputNumber * inputNumber * inputNumber)
ENDPROC
MAP cubed(anyNumber) to y
```

PROC RETURN Statement

A PROC RETURN statement causes a procedure to return control to the point immediately after the point from which the procedure was invoked (see ObjectSpeak Statement). No further statements in the procedure are executed. If an expression is included on the PROC RETURN, it is the return value of the procedure as a function call.

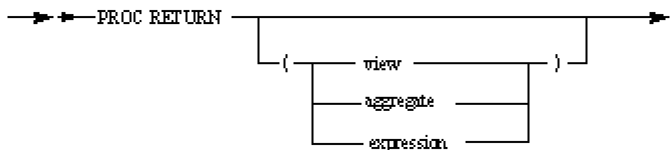
PROC RETURN Syntax

proc_return_statement:

PROC RETURN ['(' return_value ')']

return_value:

one of view aggregate expression



where:

- expression is a valid expression---see Expression Syntax.
- view is a valid view---see View.

Macros

A *macro* is a mechanism for replacing one text string in the rule source with another one. When a rule is prepared, every word in the source is scanned character by character, regardless of the structure, even the string literals are not processed as a special case. Only the special quoted strings (see Using Quoted Strings in Macros) and the Rules Language comments (see *Chapter 8. Control Statements* , Comment Statement) are not scanned. If the word is recognized as the name of a macro, it is replaced by the macro definition. This replacement is called *macro expansion*
.
The following topics about macros are described in this chapter:

- Defining Macros
- Usage of Macros
- Macro Expansion
- Using Conditional Macros
- Option Settings for Macros
- Predefined Macros

Defining Macros

To define a macro, use a definition statement of the following format:

CG_DEFINE Syntax

cg_define_statement:

CG_DEFINE '(' *macro_name* [, *string*] ')'

```
►►─CG_DEFINE(─macro_name─┬────────────┬─)──────────►◄
                         └─,─ string ─┘
```

where:

- *macro_name* is any sequence of letters, digits, and the underscore character (_) where the first character is not a digit. Macro names are case-sensitive, for example, "INIT" represents a different macro than "init." See Case-sensitivity for exceptions. Macro names cannot contain DBCS characters. If DBCS characters are used, an error is generated during the Rule preparation.
- *string* is any sequence of characters allowed in the Rules Language. The replacement string is not enclosed in quotation marks. If quotation marks are included, they are part of the replacement string and are included when the replacement string is substituted for the macro name.

CG_DEFINE must be entered in capital letters, as shown in the syntax. Blanks are not permitted between the macro function definition and the left bracket.

For example:

```
CG_DEFINE( init, MAP 0 TO HPS_WINDOW_STATE OF HPS_SET_MINMAX_I)
```

Macros can be defined anywhere, but the definition must occur before any use of the macro. Macro expansion occurs automatically during rule preparation. Use RuleView to see the results of macro expansion.

A comma used to separate any two operands in a macro statement must immediately follow the first operand with no intervening spaces. Spaces are permitted after the comma and before the second operand.

Since source is scanned for substitution regardless of the structure and macro names in CG_DEFINE are also substituted, if you do not want the macro name to be substituted, place the macro name in special quotes (see Using Quoted Strings in Macros).

Example: Macro Name Substitution in CG_DEFINE

The following example uses PRINT statement when generating C, and TRACE statement when generating Java:

```
CG_DEFINE(LANGUAGE, C)

CG_DEFINE(PRINTTRACE,
  CG_IF(<:LANGUAGE:>,Java)
      TRACE( "JAVA")
  CG_ENDIF
  CG_IF(<:LANGUAGE:>,C)
      PRINT "C"
  CG_ENDIF
)


PRINTTRACE // It is substituted with PRINT "C"
CG_DEFINE(<:LANGUAGE:>, C)


PRINTTRACE // It is substituted with PRINT "C"
CG_DEFINE(<:LANGUAGE:>, Java)

PRINTTRACE // It is substituted with TRACE("JAVA")
```

If quotes are not used in CG_DEFINE(<:LANGUAGE:>, C), infinite recursion takes place during macro expansion because the statement after LANGUAGE macro substitution looks like CG_DEFINE(C, C).

Defining Macros in Rules

Macros can be defined for a rule in two ways:

1. Use CG_DEFINE in the rule source code. The resulting definition is local and available only for the rule where the macro is defined.

2. Use one of two possible ways in the Hps.ini file. Macros defined in the Hps.ini file are available to all rules.

- Define macros in the [MacroDefinitions] section. This section defines macros that can be used for all target languages and platforms. The [MacroDefinitions] section can be viewed and updated from the Construction Workbench menu, *Tools* > *Workbench Options* > *Preparation* tab > *Conditionals* button.

This generates the following section in the Hps.ini file:

[MacroDefinitions]

macro_name_1=macro value 2
CONVERSE_CLIENT=TRUE

- Define macros in the platform/language specific section of the Hps.ini file. Two macros are defined in several sections, therefore, when the Rule is prepared, the value is chosen depending on the target platform and language. Do not change these macros. For example:

[CGen]

MACRO=LANGUAGE=C
MACRO=ENVIRONMENT=GUI

[JavaGen]

MACRO=LANGUAGE=Java
MACRO=ENVIRONMENT=GUI

[JavaBatchGen]

*MACRO=LANGUAGE=Java*MACRO=ENVIRONMENT=Batch

[JavaServerGen]

MACRO=LANGUAGE=Java
MACRO=ENVIRONMENT=Server

[JavaHTMLGen]

MACRO=LANGUAGE=Java
MACRO=ENVIRONMENT=HTML

To define a new macro, write a macro definition in an appropriate section using the following format:

MACRO=my_macro_name=my_macro_value

Using Parameters

A macro can have any number of parameters. The usage of parameters is important because recurring lines of code most frequently recur with some variations. Using parameters allow variations and still use macros for repetitive tasks.
To define a macro parameter, use the following string in the macro definition: $ *n* , where *n* is a number. The first parameter is $1, the second $2, and so on. $0 is a special case that expands to the name of the macro being defined.

The following example uses two parameters; $1 and $2:

```
CG_DEFINE( set_cursor_field,
MAP $1 TO FIELD_LONG_NAME OF SET_CURSOR_FIELD_I
MAP $2 TO VIEW_LONG_NAME OF SET_CURSOR_FIELD_I
USE COMPONENT SET_CURSOR_FIELD)*
```

To use this macro, specify the parameters enclosed within parentheses and separated by a comma:

set_cursor_field('NAME_FIELD','CUSTOMER_VIEW')

The above statement becomes (after substitution):

```
MAP NAME_FIELD TO FIELD_LONG_NAME OF SET_CURSOR_FIELD_I
MAP CUSTOMER_VIEW TO VIEW_LONG_NAME OF SET_CURSOR_FIELD_I
USE COMPONENT SET_CURSOR_FIELD)
```

The parenthesis () must be placed immediately after the macro_name with no space in-between. This means that:

macro_name (1,2)

is not a use of macro_name with two parameters (1,2) because there is a space between macro_name and (1,2). This statement is, instead, a use of macro_name with no parameter.

There is no such thing as an empty parameter list. Thus:

macro_name()

is really the macro_name used with one parameter, an empty string.

Parameters are evaluated before the macro is called. Parameter lists are variable in length and if more parameters are expected than are provided, the extra ones are taken to be the empty string.

Spaces in parameter lists are significant after a parameter. For example, if you define the following macro to join two strings:

CG_DEFINE(CAT, $1$2)

then:

"##CAT(a, b)##" expands to "##ab ##"
"##CAT(a, b)##" expands to "##a b ##"
The first space after "a" in "##a b ##" comes from the first parameter, and the space after "b" comes from the second parameter.

Example: Using Parameters in a Macro

The following example involves two invocations of the SET_PUSH_COLOR system component to set the color of the text in two push buttons. Both invocations are repetitious. The only variants are the system identifier (HPS ID) of the push button and the color specified for each.

```
MAP 'OK' TO PUSH_TEST OF SET_PUSH_COLOR_I
MAP TEXT IN WINDOW_OBJECT_PROPERTIES TO PUSH_ATTR OF SET_PUSH_COLOR_I
MAP GREEN in WINDOW_OBJECT_COLORS TO ATTR_COLOR OF SET_PUSH_COLOR_I
USE COMPONENT SET_PUSH_COLOR


IF ( RETURN_CODE OF SET_PUSH_COLOR_O <> 0)
   USE RULE MY_REPORT_COMPONENT_ERROR
ENDIF

MAP 'Cancel' TO PUSH_TEST OF SET_PUSH_COLOR_I
MAP TEXT in WINDOW_OBJECT_PROPERTIES TO PUSH_ATTR OF SET_PUSH_COLOR_I
MAP RED in WINDOW_OBJECT_COLORS TO ATTR_COLOR OF SET_PUSH_COLOR_I
USE COMPONENT SET_PUSH_COLOR

IF ( RETURN_CODE OF SET_PUSH_COLOR_O <> 0)
   USE RULE MY_REPORT_COMPONENT_ERROR
ENDIF
```

If you define a macro using a parameter in place of the two different features, system identifier (HPS ID) and color:

```
CG_DEFINE( set_pushtext,
  MAP $1 TO PUSH_TEST OF SET_PUSH_COLOR_IMAP TEXT in WINDOW_OBJECT_PROPERTIES TO PUSH_ATTR OF
SET_PUSH_COLOR_I
  MAP $2 in   WINDOW_OBJECT_COLORS TO ATTR_COLOR OF SET_PUSH_COLOR_I
  USE COMPONENT SET_PUSH_COLOR
  IF ( RETURN_CODE OF SET_PUSH_COLOR_O <> 0)
     USE RULE MY_REPORT_COMPONENT_ERROR
  ENDIF
)
```

then you can use the macro in place of the repetitious code, as follows:

```
set_pushtext( 'OK', GREEN)
set_pushtext( 'Cancel', RED)
```

Using Special Parameters in Macros

The following are special (wild card) parameters that expand to cover a range of values.

- Number of Parameters - $#
- List of All Parameters - $*
- List of All Parameters Quoted - $@
- Copied Parameters - $ and any character (except #, *, @)

Number of Parameters

$# expands to the number of parameters provided when the macro is called. In the following macro definition:

CG_DEFINE(HowMany, $#)

the following expansions result:

Macro use	Result
HowMany	*0*
HowMany()	*1*
HowMany(q,r)	*2*

List of All Parameters

*$** expands to a list of all the provided parameters with commas between. In the following macro definition:

CG_DEFINE(All, Parameters are $)*

"All(one, two)" expands to *"Parameters are one,two".*

List of All Parameters Quoted

*$@* is the same as *$** , except that all the parameters are quoted. This is quite subtle, as the process of rescanning normally removes these quotation marks again.

Copied Parameters

A "$" sign that is not followed by a digit, "#", "*" or "@" is simply copied. In the following macro definition:

CG_DEFINE(amount, $$$$)

amount

expands to:

$$$$

Undefining a Macro

Sometimes a macro needs to be undefined, for example, to prevent macro expansion within a sequence of text containing the macro. To undefine a macro, use a statement of the following form:

CG_UNDEFINE Syntax

cg_undefine_statement:

CG_UNDEFINE '(' *macro_name* ')'

CG_UNDEFINE(macro_name)

Assuming the macro is already defined, you need to use quotes to prevent macro expansion within the *CG_UNDEFINE* statement. For example:

```
CG_DEFINE( map, <:MAP $1 TO C_TEXT:>)
CG_UNDEFINE( <:map:>)
map
```

This results in "map" being undefined, so the string "map" is copied into the rule source (no substitution is performed).

Usage of Macros

The following topics describe some of the common usages of macros:

- Declaring Constants
- Substituting Rule Source Code
- Embedding Macros

Declaring Constants

One common use of macros is to declare constants. For example, you can use the following macro definition to declare a constant for the value of pi:

CG_DEFINE(pi, 3.14159)

Once the constant is defined, you can use it in any subsequent statement, such as:

MAP pi * (RADIUS2) TO AREA

Using a macro to declare a constant is good programming practice because it allows you to use a meaningful name for the constant rather than a number. Using a constant is preferable to using a variable because a constant allows the compiler to optimize statements referencing the constant.

Substituting Rule Source Code

The most common and powerful use of macros is to use them in place of the Rule source code. In applications where lines of code are repeated frequently, you can define a macro and use the macro name in your code. When you prepare the Rule, AppBuilder replaces the macro name with the actual lines of code.

Example: Using a Macro for Rule source code

For example, suppose that after each use of a system component, you check the return code in the same way and invoke the same error routine in case of an error:

```
USE COMPONENT SET_PUSH_COLOR
IF ( RETURN_CODE OF SET_PUSH_COLOR_O <> 0)
    USE RULE MY_REPORT_COMPONENT_ERROR
ENDIF
```

If you define a macro as:

```
CG_DEFINE( check_return,
IF ( RETURN_CODE of SET_PUSH_COLOR_O <> 0)
    USE RULE MY_REPORT_COMPONENT_ERROR
ENDIF
)
```

then after every use of the system component, you can check the return code by using the macro:

```
USE COMPONENT SET_PUSH_COLOR
check_return
```

Embedding Macros

Macro expansion takes place wherever the macro occurs. Because macro processing is done before code generation, macros are not subject to the usual Rules Language constraints. This means that you can even use a macro inside a string. For example:

```
CG_DEFINE(LISTSIZE, 10)
MAP 'Use LISTSIZE entries' TO F_PROMPT
```

results in:

MAP 'Use 10 entries' TO F_PROMPT

Macros can also be used inside other macro definitions. For example:

```
CG_DEFINE(LISTSIZE, 10)
CG_DEFINE( SETPROMPT, MAP 'Use LISTSIZE entries' TO F_PROMPT)
SETPROMPT
```

also results in:

MAP 'Use 10 entries' TO F_PROMPT

Macro Expansion

Every identifier that is not in comments or in special quoted strings (see Using Quoted Strings in Macros) is looked up in a dictionary. If this identifier is equal to a previously-defined macro name, it is expanded (substituted) with its macro value. Unlike most Rules Language processing, the lookup is case-sensitive. Because expansion is done before the code generation, it must result in the valid Rules Language code or an error occurs during the code generation.
The same is true for string literals; however, only the whole identifier is looked up, not any part of it. For example:

CG_DEFINE(RULE_NAME, my_rule)

The following will be substituted:

- 'RULE_NAME'
- "RULE_NAME"
- RULE_NAME%

  Percent sign is a special symbol, thus it is not a part of the identifier. RULE_NAME will be substituted and result string will be my_rule%

No substitution occurs in the following strings:

- MY_RULE_NAME

  In this case, MY_RULE_NAME is not known as a macro name, RULE_NAME is in the middle of the identifier.
- "RULE_NAME_1"

  This is also the case where RULE_NAME is only a part of another identifier RULE_NAME_1.
- _RULE_NAME

  Underscore is a part of the identifier; this is not a special symbol, so this will not be substituted either.

Using Quoted Strings in Macros

There may be times when you do not want macro expansion to be performed on a string. To prevent the macro expansion (substitution), enclose the string within special quotation marks: *<:* at the beginning and *:>* at the end. These special quotation marks can be nested. One level of quotes

is stripped off as the rule is processed. Thus <:::> becomes an empty string and <:<:quantity:>:> becomes <:quantity:>.

It is commonplace to quote a macro's definition to prevent expansion:

CG_DEFINE(macroname, <:This is quoted to prevent expansion:>)

To change the quotation marks, see Changing the Quote Characters in Macros.

Example: Using Quoted Strings in a Macro

Suppose you want to map both the name of a macro and its definition into a character variable. The following statements:

```
CG_DEFINE( quantity, 16)
MAP 'quantity is quantity' TO C_TEXT
```

result in the clearly inappropriate statement:

MAP '16 is 16' TO C_TEXT

To prevent the macro expansion, quote the first word "quantity":

```
CG_DEFINE( quantity, 16)
MAP '<:quantity:> is quantity' TO C_TEXT
```

this results in the statement:

MAP 'quantity is 16' TO C_TEXT

Quote strings are recognized anywhere; therefore:

```
CG_DEFINE( quantity, 16)
MAP 'qu<:ant:>ity is quantity' TO C_TEXT
```

achieves the same result.

Changing the Quote Characters in Macros

The default quote strings are <: to start the quotes and :> to finish them. These quote characters are unlikely to clash with normal Rules Language text. However, they can be changed at any time with a CG_CHANGEQUOTE statement.

CG_CHANGEQUOTE Syntax

cg_changequote_statement:

CG_CHANGEQUOTE ‘(‘ *open*, *close* ‘)’

CG_CHANGEQUOTE(*open*, *close*)

where:

- *open* is the string to start the quotes.
- *close* is the string to end the quotes.

For example, to use ' (single quotation mark) to open and to close quote strings, use the macro statement:

CG_CHANGEQUOTE(', ')

If one of the parameters is missing, the default <: and :> are used instead.

To disable the quoting mechanism entirely, change the quotation marks to empty strings.

CG_CHANGEQUOTE(,)

A limitation of macro definitions is that there is no way to quote a string containing an unmatched left quotation mark. You can circumvent this by disabling quoting temporarily, then reinstating it.

Quote strings should never start with a letter or an underscore (_).

Recursive Macro Expansions

Normally, the macro preprocessor scans the rule source once. However, when a macro expansion (substitution) takes place, the result is rescanned again so that further substitutions can be made. For example:

```
CG_DEFINE( red , 6)
CG_DEFINE( turnred, <:MAP red TO BUTTON:>)
turnred
```

The macro "turnred" uses the quotes to prevent the substitution for "red" before it is used. When "turnred" is used, its substitution is "MAP red TO BUTTON", but the subsequent rescanning spots the macro "red" and substitutes it with "6". The result of this example becomes as follows:

MAP 6 TO BUTTON

Rescanning can lead to a problem of infinite recursion. Care must be taken to avoid this. Consider:

```
CG_DEFINE( infinite, <:infinite infinite:>)
infinite
```

When the macro "infinite" is expanded, its expansion is two copies of itself, which are then expanded again (due to rescanning), and so on. There is no defined limit to the depth of such recursion. This can be prevented by using nested quotes:

```
CG_DEFINE( infinite, <:<:infinite infinite:>:>)
infinite
```

The extra level of quotes prevents the rescan from seeing the recursive use of infinite, and the above example results in:

infinite infinite

Using Conditional Macros

Several macro statements test conditions to allow decisions to be taken and alternate expansions selected. For example, by using conditional macros, it is possible to create one rule source code that can be used for both C and Java platforms.
This section describes the usage of conditional macros:

- Evaluating if a Macro Exists
- Comparing Values
- Using Conditional Translation

Evaluating if a Macro Exists

The following macros cause different expansions depending on whether a particular macro is defined or not:

CG_IFDEF Syntax

CG_IFDEF(macro_name , string [, string])

where:

- *string* is any sequence of characters allowed in the Rules Language.

When only one *string* is specified, a macro expansion takes place if *macro_name* exists.
When two *strings* are specified, and if the *macro_name* does not exist, the second *string* is substituted.

Example: Using CG_IFDEF

Example 1: Using one string

In the following example, note that the use of quoting prevents the macro from being expanded:

```
CG_DEFINE( DB2)
CG_IFDEF( <:DB2:>, MAP '<:DB2:>' TO DB)
```

This results in:

MAP 'DB2' TO DB

Example 2: Using two strings

Two strings are used in the following CG_IFDEF statement:

```
CG_UNDEFINE( <:DB2:>)
CG_IFDEF( <:DB2:>, MAP 'DB2' TO DB, MAP 'Default' TO DB)
```

Because the macro DB2 is undefined, it no longer exists, thus the second *string* is used. This results in:

MAP 'Default' TO DB

Comparing Values

The following macro compares values and performs substitution based on the result of the comparison:

CG_IFELSE Syntax

cg_ifdef_statement:

CG_IFDEF '(' *macro_name*, *string* [, *string*] ')'

where:

- *value1* is the first value used in the comparison, and is typically a macro name.
- *value2* is the second value used in the comparison.
- *string* is any sequence of characters allowed in the Rules Language.

When the CG_IFELSE statement has three parameters (*value1* , *value2* and one *string*), the *value1* is compared with the *value2* for string equality, and if they are equal, it substitutes the third parameter, the *string* .

When the CG_IFELSE statement has four parameters (*value1* , *value2* and two *strings*), the *value1* is compared with the *value2* for string equality, and if they are equal, it substitutes the third parameter; otherwise it substitutes the fourth parameter.

If more than four parameters are passed, then the *value1* and the *value2* are compared for string equality, and if they are equal, it substitutes the first *string* ; if they are not equal, the first three parameters are stripped, and the process repeats until no parameters are left. You can use more than four parameters to code a CASE-OF statement.

For example, to write one source code to define the Enabled property of a push button object in both C and Java, you can write a conditional macro as follows:

```
CG_IFELSE(LANGUAGE, C,
ExitButton.Enabled(1),
MAP TRUE TO ExitButton.Enabled)
```

In the above example, LANGUAGE is a predefined macro whose value is set to the platform where the rule is being prepared. Thus, for C, preparing the above rule results in:

ExitButton.Enabled(1)

For Java (prepared using "else" code), the result is:

MAP TRUE TO ExitButton.Enabled

Since objects are not supported in COBOL, the above example works well.

Example: Using CG_IFELSE

Example 1: CG_IFELSE statement with three parameters

```
CG_DEFINE( DB, DB2)
CG_IFELSE( DB, DB2, <:MAP 'DB2' TO DBASE:>)
```

In the above example, the first parameter DB is expanded to DB2, which is equal to the second parameter, this results in:

```
MAP 'DB2' TO DBASE
CG_DEFINE( DB, Sybase)
CG_IFELSE( DB, DB2, <:MAP 'DB2' TO DBASE:>)
```

The above example results in nothing because the first parameter DB is expanded to Sybase, which is not equal to the second parameter.

Example 2: CG_IFELSE statement with four parameters

```
CG_DEFINE( DB, DB2)
CG_IFELSE( DB, DB2, <:MAP 'DB2' TO DBASE:>, <:MAP 'Sybase' TO DBASE:>)
```

The first and the second parameters are equal, thus the above example results in:

```
MAP 'DB2' TO DB
CG_DEFINE( DB, Sybase)
CG_IFELSE( DB, DB2, <MAP 'DB2' TO DBASE:>, <:MAP 'Sybase' TO DBASE:>)
```

The first and the second parameters are not equal, thus the above example results in:

MAP 'Sybase' TO DB

Example 3: CG_IFELSE statement with more than four parameters

```
CG_DEFINE( DB, Oracle)
CG_IFELSE( DB, <:Oracle:>, <:MAP 'Oracle' TO DBASE:>,
DB, <:Sybase:>, <:MAP 'Sybase' TO DBASE:>,
DB, <:Informix:>, <:MAP 'Informix' TO DBASE:>,
DB, <:DB2:>, <:MAP 'DB2' TO DBASE:>, <:MAP 'Other' TO DBASE:>)
```

The first parameter expands to Oracle, thus this results in:

MAP 'Oracle' TO DBASE

```
CG_DEFINE( DB, DB2)
CG_IFELSE( DB, <:Oracle:>, <:MAP 'Oracle' TO DBASE:>,
DB, <:Sybase:>, <:MAP 'Sybase' TO DBASE:>,
DB, <:Informix:>, <:MAP 'Informix' TO DBASE:>,
DB, <:DB2:>, <:MAP 'DB2' TO DBASE:>, <:MAP 'Other' TO DBASE:>)
```

The first parameter expands to DB2, thus this results in:

MAP 'DB2' TO DBASE

```
CG_UNDEFINE( <:DB:>)
CG_IFELSE( DB, <:Oracle:>, <:MAP 'Oracle' TO DBASE:>,
DB, <:Sybase:>, <:MAP 'Sybase' TO DBASE:>,
DB, <:Informix:>, <:MAP 'Informix' TO DBASE:>,
DB, <:DB2:>, <:MAP 'DB2' TO DBASE:>, <:MAP 'Other' TO DBASE:>)
```

Because the first parameter DB is undefined, it no longer equals to any of the values, thus this results in:

MAP 'Other' TO DBASE

Using Conditional Translation

This topic describes the following conditional translations:

- CG_IF Statement with Condition Based on Defined Macro Name
- CG_CASEOF Statement
- CG_IF Statement with Boolean Condition

CG_IF Statement with Condition Based on Defined Macro Name

In the following syntax drawing, with the CG_IF and CG_IFNOT statements, th *e macro name* is compared with the *value* . When using CG_IF, if the *macro_name* and the *value* are equal, all *statements* after CG_ELSE are excluded from translation; if the *macro_name* and the *value* are not equal, only *statements* after CG_ELSE are processed. CG_IFNOT works completely the other way: all *statements* after CG_ELSE are excluded from translation if the *macro_name* and the *value* are not equal.

Macro IF Syntax

macro_if_statement:

cg_if_part (cg_elseif_part)* [cg_else_part] CG_ENDIF

cg_if_part:

CG_IF '(' *macro_name*, *value* ')' statement_list

CG_IFNOT '(' *macro_name*, *value* ')' statement_list

CG_IFDEFINED '(' *macro_name* ')' statement_list

CG_IFNOTDEFINED '(' *macro_name* ')' statement_list

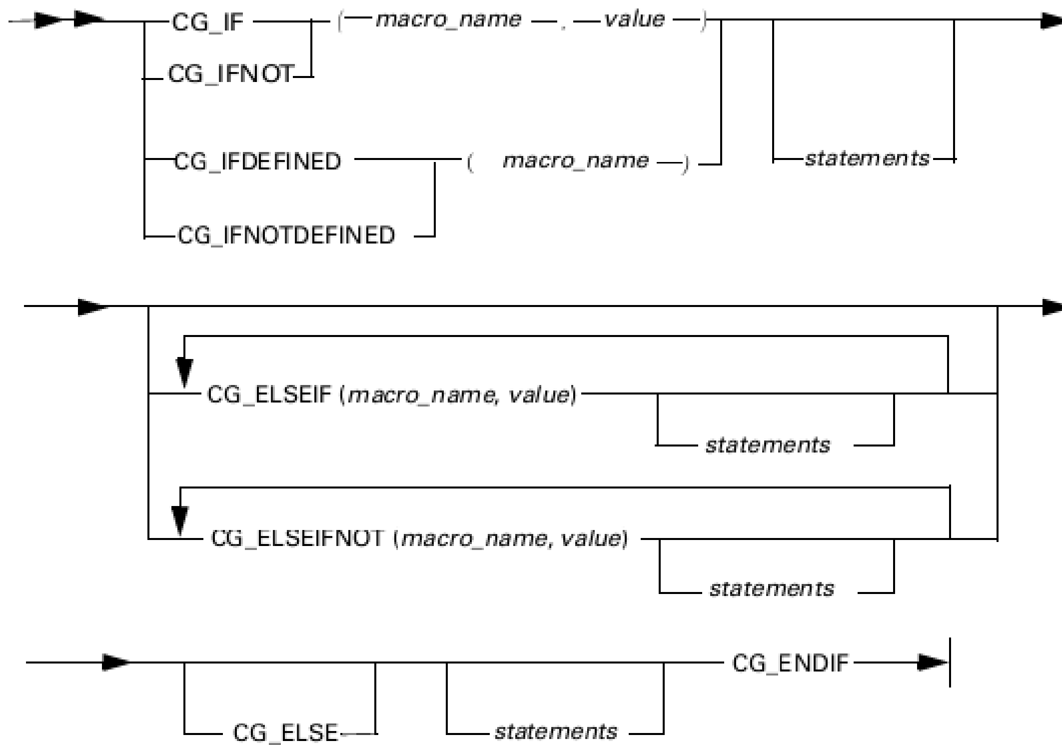
cg_elseif_part:

CG_ELSEIF '(' *macro_name*, *value* ')' statement_list

CG_ELSEIFNOT '(' *macro_name*, *value* ')' statement_list

cg_else_part:

CG_ELSE statement_list



where:

- *macro_name* is any macro name.
- *value* is any string that could be assigned to *macro_name.*
- *statements* are any Rules Language statements.

Do not place extra spaces after preprocessor command parameters because extra space is considered to be a part of a parameter. Spaces are allowed just before parameters. For example:

```
CG_DEFINE(JAVA, TRUE)
CG_IF(JAVA , TRUE) *>Additional space after 1st parameter<*
*>and before second one<*
```

Because of the extra space after the first CG_IF parameter, it is treated as "JAVA_" (underscore means space) and expanded into "TRUE_". The same logic applies for spaces before other parameters.

Example: Using CG_IF and CG_IFNOT

```
CG_DEFINE(database, Informix)
CG_IF(database, Informix)
Map 1 to i
CG_ELSE
Map 0 to i
CG_ENDIF
```

Because *database* was defined as *Informix* , "Map 1 to i" is processed.

The same result can be achieved using CG_IFNOT:

```
CG_DEFINE(database, Informix)
CG_IFNOT(database, Informix)
Map 0 to i
CG_ELSE
Map 1 to i
CG_ENDIF
```

CG_IFDEFINED and CG_IFNOTDEFINED

In the CG_IFDEFINED(*macro_name*) statement, the preprocessor analyzes to determine if *macro_name* has been defined. If it has been defined, all *statements* after CG_ELSE are excluded from translation; if it has not been defined, only *statements* after CG_ELSE are processed.

In the CG_IFNOTDEFINED(*macro_name*) statement, if the *macro_name* has not been defined, all *statements* after CG_ELSE are excluded from translation; if it has been defined, only *statements* after CG_ELSE are processed.

Note that the parameter (*macro_name*) used in the *CG_IFDEFINED* statement and the *CG_IFNOTDEFINED* statement is expanded (as for all other commands) unless it is placed in *CGMEX* quotation marks (*<:* and *:>* by default).
In this example:

```
CG_DEFINE(JAVA, TRUE)
CG_IFDEFINED(JAVA)
Map 1 to i
CG_ELSE
Map 0 to i
CG_ENDIF
```

"Map 0 to I" is processed because the CG_IFDEFINED parameter is expanded into TRUE but the macro named TRUE is not defined. But if you were to place JAVA in quotation marks, as in the example: (*CG_IFDEFINED(<:JAVA:>))* , "Map 1 to I" is processed because <:JAVA:> is expanded into JAVA and this macro is defined.

Example: Using CG_IFDEFINED and CG_IFNOTDEFINED

```
CG_UNDEFINE(Java)
CG_IFDEFINED(<:Java:>)
Map 1 to i
CG_ELSE
Map 0 to i
CG_ENDIF
```

Because Java was undefined, "Map 0 to i" is processed.

```
CG_UNDEFINE(Java)
CG_IFNOTDEFINED(Java)
   Map 1 to i
CG_ELSE
  Map 0 to i
CG_ENDIF
```

Because Java was undefined, "Map 1 to i" is processed.

CG_ELSEIF and CG_ELSEIFNOT

You can insert multiple CG_ELSEIF and CG_ELSEIFNOT statements to evaluate more conditions in one CG_IF statement.

After the CG_IF statement evaluates to false, the CG_ELSEIF(*macro_name, value*) statement compares th *e macro_name* with the *value* , and if they are equal, all *statements* after CG_ELSEIF are processed. In the CG_ELSEIFNOT(*macro_name, value*) statement, th *e macro_name* is compared with the *value* , and if they are not equal, all *statements* after CG_ELSEIFNOT are processed.

Example: Using CG_ELSEIF

```
CG_IF(Language,C)
  Map "c" to sLang
CG_ELSEIF(Language,Cobol)
  Map "Cobol" to sLang
CG_ELSEIF(Language,Java)
  Map "Java" to sLang
CG_ELSE
  CG_CGEXIT(8) //unsupported language
CG_ENDIF
```

CG_CASEOF Statement

The CG_CASEOF macro statement switches translation between any number of groups of statements, depending on the result of comparing *macro_name* with the *values* for each group.

Macro CG_CASEOF Syntax

cg_caseof_statement:

CG_CASEOF ‘(‘ *macro_name* ‘)’ cg_switches_part CG_ENDCASEOF

cg_switches_part:

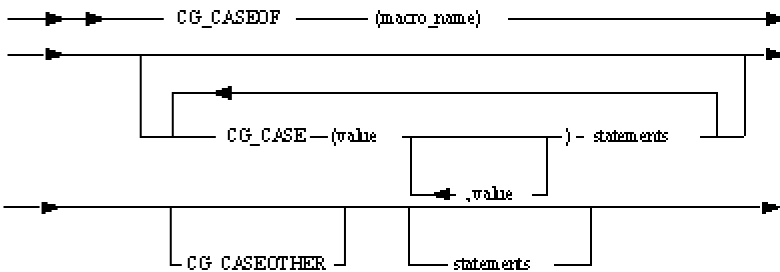
(cg_case_part)* [cg_caseother_part]

cg_case_part:

CG_CASE ‘(‘ *value* (, *value*)* ‘)’ statement_list

cg_caseother_part:

CG_CASEOTHER statement_list



where:

- macro_name is any macro name.
- value is any string that could be assigned to macro_name.
- statements are any Rules Language statements.

A CG_CASEOF statement determines which one, if any, of values in the subordinate CG_CASE clauses equals the macro_name in the CG_CASEOF clause. Translation continues with the statements following that CG_CASE clause.

Example: Using CG_CASEOF

```
CG_DEFINE (TARGET, Java)
CG_CASEOF (TARGET)
CG_CASE (Java)
trace('Target language is Java)
CG_CASE (Cobol, OpenCobol)
trace('Target language is COBOL)
CG_ENDCASEOF
```

Result of macro expansion will be:

trace('Target language is Java')

If none of the CG_CASE clauses has value equal to the macro_name translation continues with the statements following the optional CG_CASEOTHER.

Example: Using CG_CASEOF and CG_CASEOTHER

```
CG_DEFINE (TARGET, Java)
CG_CASEOF (TARGET)
CG_CASE (C)
trace('Target language is C')
CG_CASE (Cobol, OpenCobol)
trace('Target language is COBOL')
CG_CASEOTHER
trace('Target language is neither C nor any Cobol dialect')
CG_ENDCASEOF
```

Result of macro expansion will be:

trace('Target language is neither C nor any Cobol dialect')

If none of CG_CASE clauses has value equal to the macro_name and there is no CG_CASEOTHER clause, no statements are included into translation.

Example: Using CG_CASEOF without CG_CASEOTHER clause and no statements

```
CG_DEFINE (TARGET, Java)
CG_CASEOF (TARGET)
CG_CASE (C)
trace('Target language is C')
CG_CASE (Cobol, OpenCobol)
trace('Target language is some Cobol dialect')
CG_ENDCASEOF
```

Result of macro expansion will be empty.

CG_IF Statement with Boolean Condition

Macro CG_IF with Boolean Condition Syntax

macro_boolean_if_statement:

cg_if_part (cg_elseif_part)* [cg_else_part] CG_ENDIF

cg_if_part:

CG_IF '(' condition ')' statement_list

cg_elseif_part:

CG_ELSEIF '(' condition ')' statement_list

cg_else_part:

CG_ELSE statement_list

condition:

macro_name = *value*

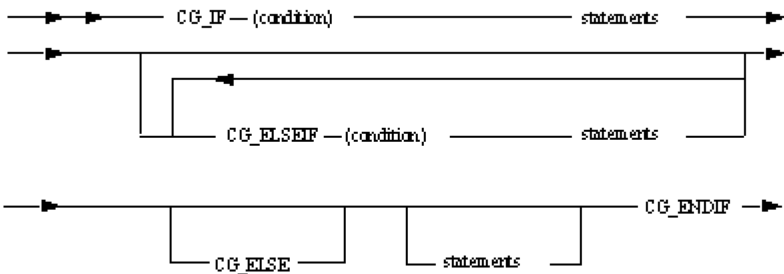
macro_name IS DEFINED

condition AND condition

condition OR condition

NOT condition

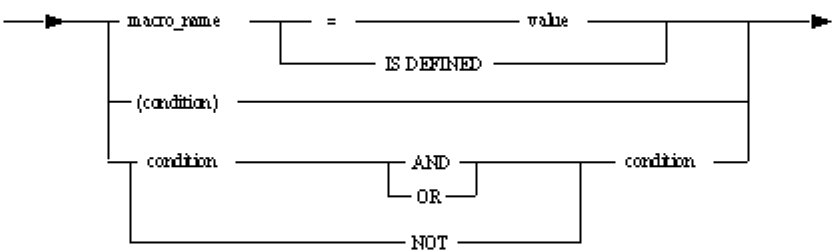
The following macro switches the translation depending of the truth or falsity of a condition.



where:

- condition is condition expression.
- statements are any Rules Language statements.

Condition Syntax



where:

- macro_name is any macro name.
- value is any string that could be assigned to macro_name. It is recommended to use special quotes <^ for begin, and ^> for end value, if value contains more then one word and it is necessary to use this quotes when value contains GCMEX key-words or non-balanced brackets.

Processing of CG_IF, CG_ELSEIF and CG_ELSE – just like the CG_IF statement with comparison of macro_name and some value. Actually this is more common case of the CG_IF statement representation.

Example: Using of CG_ELSE and CG_ELSEIF clauses

Example 1: Using CG_ELSE clause

```
CG_DEFINE (TARGET, Cobol)
CG_IF (<:TARGET:> IS DEFINED AND
(TARGET = OpenCobol OR TARGET = Cobol))
trace('Target language is some Cobol dialect')
CG_ELSE
trace('Target language is not any Cobol dialect')
CG_ENDIF
```

Result of macro expansion will be:

trace ('Target language is some Cobol dialect')

Example 2: Using of CG_ELSEIF clause

```
CG_DEFINE (TARGET, C AND C++)
CG_IF (<:TARGET:> IS DEFINED AND
(TARGET = OpenCobol OR TARGET = Cobol))
trace('Target language is some Cobol dialect')
CG_ELSEIF (TARGET= <^C AND C+\+^>)
trace ('Target language is some C dialect')
CG_ELSE
trace('Target language is TARGET')
CG_ENDIF
```

Result of macro expansion will be:

trace ('Target language is undefined')

Option Settings for Macros

There are several optional settings that can be used with macros to provide flexibility to the general rule. These options are only supported for CG_IF.

- Case-sensitivity
- Macro Name Validation

Case-sensitivity

The case-sensitivity configuration option (code generation parameter) controls the case-sensitivity of the macro preprocessor. Only CG_IF is supported for this option. All macro values are case-sensitive, and by default, case-sensitivity is not specified to facilitate backwards compatibility.

When this option is specified, the preprocessor ignores the case of the macro value.
For example, assume the following section exists in the Hps.ini file and the rule has been prepared for Java:

[JavaGen]

MACRO=LANGUAGE=Java
MACRO=ENVIRONMENT=GUI

When the option is specified, the following statement is evaluated as TRUE:

```
CG_IF(LANGUAGE, JAVA)
Set x := 1
CG_ELSE
Set x := 2
CG_ENDIF
```

and the statement " *Set x := 1* " will be left in the output file.

When the option is not specified, the value of the condition is evaluated as FALSE, and the statement " *Set x := 2* " will be left in the output file. Also, the listing contains the warning:

52954-W Case-insensitive comparison is true for macro "LANGUAGE"

XXXXX-W Macro LANGUAGE was defined as "Java", comparison failed.

Since the case of possible values of the LANGUAGE macro listed in the [MacroDomains] section of the Hps.ini file is already taken into account, the listing also contains the following error:

ERROR:

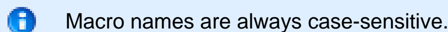
52960-S Value "JAVA" is not listed in the domain for macro "LANGUAGE"

To disable case-sensitivity, use the following code generation parameter:

-fMEXCI

The following flag in the Hps.ini file can also be used:

FLAG=MEXCI



Macro names are always case-sensitive.

Macro Name Validation

Even if a macro is not defined, you can use the macro in a rule, but a warning is generated when the rule is prepared. The macro name validation configuration option controls what is generated when a macro name cannot be validated. Only CG_IF is supported for this option.

For example, if there is no definition for a macro named TARGET_LANGUAGE, then the statement

```
CG_IF(TARGET_LANGUAGE, German)
Set PUSH_TEXT of SET_PUSH_MODE_I := 'Speichern'
CG_ENDIF
```

will not produce any code in the output file.

By default, this option is disabled, and the following warning is generated:

52825-W Undefined macro "TARGET_LANGUAGE".

When this option is enabled, the rule preparation fails with the following error message:

52965-S Macro "TARGET_LANGUAGE" is not defined.

This option works in the same way for macro definitions that have domains defined. See Validating Macros in Domain

To validate macro names, use the following code generation parameter:

-FMMBDEF

The following flag in the Hps.ini file can also be used:

FLAG=MMBDEF

Validating Macros in Domain

If the " *macro_name* " exists in the [MacroDomains] section in the Hps.ini file, then the value used in the rule is validated against the values in the domain (the list of specified values with case-sensitivity defined using the Case-sensitivity option). If the value is not in the domain, the rule preparation fails with an error message.
For example, assume the Hps.ini file contains the following definitions:

[MacroDomains]

LANGUAGE=Java,C

[JavaGen]
MACRO=LANGUAGE=Java
MACRO=ENVIRONMENT=GUI

[CGen]
MACRO=LANGUAGE=C
MACRO=ENVIRONMENT=GUI

And the rule code contains the following statements with the case-sensitivity being disabled:

```
CG_IF(LANGUAGE, JAVA)
Set x := 1
CG_ELSE
Set x := 2
CG_ENDIF

CG_IF(ENVIRONMENT, HTML)
Set y := 1
CG_ELSE
Set y := 2
CG_ENDIF

CG_IF(LANGUAGE,Cobol)
Set z := 3
CG_ENDIF
```

The "JAVA" value is validated against the values in the domain list, but the "HTML" value is not validated.

The statement:

CG_IF(LANGUAGE,Cobol)

results in a preparation failure with the following error message:

W52960-S Value "Cobol" is not listed in the domain for macro LANGUAGE

Predefined Macros

There are two predefined macro definitions that can be used in rules.

- LANGUAGE
- ENVIRONMENT

They are used only on the PC workstation. These macros are defined in the AppBuilder initialization file (Hps.ini).
The values are set according to the platform to which they are generating code as summarized in the following table.

Values for LANGUAGE and ENVIRONMENT Macros

When generating	LANGUAGE=	ENVIRONMENT=
Java client code	Java	GUI
Java server code for RMI or EJB	Java	Server
Java servlet client code for HTML client	Java	HTML

C code for client Windows application	C	GUI
C code for server Windows application	C	Server
COBOL	Cobol	Server
CSharp client code	CSharp	GUI
CSharp server code	CSharp	Server

There are a number of predefined macros that perform specific tasks.

- Including Files
- Using Date and Time Macros
- Using Name Macros
- Using Debugging Macro
- Exiting from Translation
- Using Recursion to Implement Loops
- Using String Functions
- Using Arithmetic Macros

Including Files

The CG_INCLUDE statement causes the compiler to process the file specified in the *file_name* parameter. This file must contain allowable Rules Language statements and the *File_name* string format must be allowable on the platform where the rule is translated.

CG_INCLUDE Syntax

Macro CG_INCLUDE Syntax

cg_include_statement:

CG_INCLUDE '(' *file_name* ')'

CG_INCLUDE (*File_name*)

where:

- *File_name* is the string specifying a file name.

For example:

CG_INCLUDE (e:\include\commonrulepart.inc)

Currently the CG_INCLUDE macro expansion failure is handled differently by the host and the workstation. On the host side, when you perform rule preparation using the CG_INCLUDE (myfile) statement, and the dataset is specified neither in the codegen.ini nor in the rule source, you receive the following RC=8 error message:

ERROR: 52903-S Error reading include file 'M90SQLEP' with errno=49 (EDC5049I. The specified file name could not be located.) See also Messages Reference Guide.

However, when preparing the same rule in the same manner during Java preparation on the workstation, the system issues a warning and the preparation continues successfully.

For details about the INI settings concerning the file inclusion, see also General settings for the Codegen section, from General settings for the Codegen section.

Using Date and Time Macros

There are a number of predefined macros that manipulate date and time in various ways:

- *CG_RULE_TRANSLATION_DATE* translates the date in 'mm dd yyyy' format.
- *CG_RULE_TRANSLATION_TIME* translates the time in 'hh:mm:ss' format.
- *CG_RULE_TRANSLATION_TIMESTAMP* translates the timestamp in 'mm dd yyyy hh:mm:ss' format.
- *CG_CODEGEN_VERSION* translates the code generation version of the date and time in 'Mmm dd yyyy hh:mm:ss' format.

[Example: Using Date and Time Macros]

```
dcl
  ttime varchar(100);
enddcl
map "CG_RULE_TRANSLATION_TIMESTAMP" to ttime
```

The above statement is expanded to:

map "02 14 2004 21:13:15" to ttime

The following is a CG_CODEGEN_VERSION statement example:

```
dcl
cver varchar(100);
enddcl
map "CG_CODEGEN_VERSION" to cver
```

The above statement will be expanded to:

map "Feb 14 2004 21:13:15" to cver

Using Name Macros

The following is a list of predefined macros that manipulate Rule names:

- *CG_RULE_SHORT_NAME* translates the rule short name.
- *CG_RULE_LONG_NAME* translates the rule long name.
- *CG_RULE_IMP_NAME* translates the rule implementation name.

[Example: Using Name Macros]

```
dcl
  rshortname, rlongname varchar(100);
enddcl
map "CG_RULE_SHORT_NAME" to rshortname
map "CG_RULE_LONG_NAME" to rlongname
```

The above statements will be expanded to:

```
map "AA7FBN" to rshortname
map "MY_TEST_RULE" to rlongname
```

For the following support functions: *getRuleShortName* , *getRuleLongName* and *getRuleImpName* , the mechanism to capture the name also uses the predefined name macros. Every rule has access to these three macros:

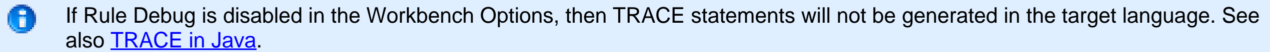
- CG_RULE_SHORT_NAME – this macro is replaced by rule's long name
- CG_RULE_LONG_NAME – this macro is replaced by rule's long name
- CG_RULE_IMP_NAME – this macro is replaced by rule's long name.

Using Debugging Macro

The predefined macro CG_DEBUG works the same as _DEBUG in Visual C++. This macro excludes the debug code from the release version and is used only for debugging purposes.
You can enable or disable CG_DEBUG macro from the Construction Workbench menu by selecting *Tools* > *Workbench Options* > *Preparation* tab, and check or uncheck Rule Debug. CG_DEBUG contains the value of TRUE when this macro is defined. This macro can be disabled by the -YD Codegen parameter (see Command Line Parameters Settings).

Example: Debugging Macro

```
CG_IF(CG_DEBUG,TRUE)
   PERFORM My_Debug_Proc()
CG_ENDIF
```



If Rule Debug is disabled in the Workbench Options, then TRACE statements will not be generated in the target language. See also TRACE in Java.

Exiting from Translation

The CG_CGEXIT statement breaks the process of translation with the return code *Return code.*

CG_EXIT Syntax

Macro CG_EXIT Syntax

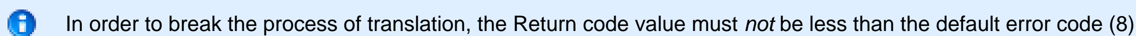
cg_exit_statement:

CG_EXIT '(' *return_code* ')'

CG_CGEXIT (*Return code*)

where:

- *Return code* is an integer number.



In order to break the process of translation, the Return code value must *not* be less than the default error code (8)

Example: Exiting from translation

```
CG_IFDEFINED(Cplusplus)
  Map 1 to i
CG_ELSE
  CG_CGEXIT(8) *> this rule created for C+\+ only <*
CG_ENDIF
```

Using Recursion to Implement Loops

Although macros do not directly support loops, it is possible to simulate the effect of looping by using recursion and conditionals. This is a very complex procedure, and you must exercise caution to avoid creating endless loops. To create loops, there is a special macro statement: CG_SHIFT. This macro uses recursion to process parameters one by one. CG_SHIFT takes any number of parameters and returns the same list (each parameter quoted) after removing the first parameter.

Example: Using CG_SHIFT to implement loops

In the following example, the macro allows many variables to be set to 0 with one simple call:

```
CG_DEFINE( clear_all,
  <:CG_IFELSE( $1, <::>, ,
    <:<:MAP 0 TO $1:>
      clear_all(CG_SHIFT($@)):>):>)
clear_all( COUNTER, HEIGHT, WIDTH)
```

This results in:

```
MAP 0 TO COUNTER
MAP 0 TO HEIGHT
MAP 0 TO WIDTH
```

Using String Functions

There are a number of predefined functions that manipulate strings in various ways:

- CG_LEN
- CG_INDEX
- CG_SUBSTR

CG_LEN

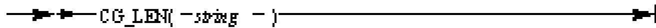
CG_LEN returns the length of a string.

Syntax

Macro CG_LEN Syntax

cg_len_statement:

CG_LEN '(' *string* ')'



For example,

CG_LEN() results in 0.
CG_LEN(<:database:>) results in 8.

CG_INDEX

CG_INDEX returns the position of the *substring* in the *string* . The search is case-sensitive. It returns 0 if the *substring* is not found in the *string* .

Syntax

Macro CG_INDEX Syntax

cg_index_statement:

CG_INDEX '(' *string*, *substring* ')'

For example,
CG_INDEX(<:DB2/2 database access:>, <:DB2:>) results in 1.
CG_INDEX(<:DB2/2 database access:>, <:Oracle:>) results in 0.

CG_SUBSTR

CG_SUBSTR extracts some part of a *string* starting at the *from* position. If *length* is specified, it is the maximum size of the string returned. If *length* is not specified, the macro returns everything to the end of the string.

Syntax

Macro CG_SUBSTR Syntax

cg_substr_statement:

CG_SUBSTR ‘(‘ *string*, *from* [, *length*] ‘)’

For example,
CG_SUBSTR(<:DB2/2 database access:>, 16) results in access.
CG_SUBSTR(<:DB2/2 database access:>, 1, 5) results in DB2/2.

Using Arithmetic Macros

Several macro statements support integer arithmetic. Integer arithmetic with macros is performed to 32-bit precision---the same as for integers in the Rules Language.

- CG_INCR and CG_DECR
- CG_EVAL

All calculations are performed using native C arithmetic that corresponds to CALCULATOR arithmetic with INTEGER type. Because native arithmetic is used, it is not possible to detect overflow situations and the result of any macro statements with an overflow is unpredictable. Nevertheless, division by zero condition is handled; if an expression contains division by zero, a compile-time error is generated.

There are several ways to express numbers to allow various radixes (number base) to be specified. See the following table:

Expressing numbers to allow various radixes to be specified

Ways of expressing numbers	Example
No prefix indicates decimal	22 49 78 23456
A single 0 indicates octal	007 02 0123
0x indicates hexadecimal	0x1ff 0x55 0xabcd
0b indicates binary	0b1101
0r (where is a decimal number from 2 to 36) indicates a specific radix	06:555 base 6 012:bbb duodecimal

To change precedence, use parentheses "(" and ")".

CG_INCR and CG_DECR

CG_INCR and CG_DECR macro statements increment or decrement an integer and return the result.

Syntax

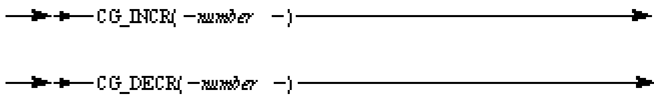
Macro CG_INCR and CG_DECR Syntax

cg_incr_statement:

CG_INCR '(' *number* ')'

cg_decr_statement:

CG_DECR '(' *number* ')'



Example: CG_INCR and CG_DECR

CG_INCR(29) results in 30.
CG_DECR(14) results in 13.

The following statements result in 11:

```
CG_DEFINE(amount, 10)
CG_INCR(amount)
```

CG_EVAL

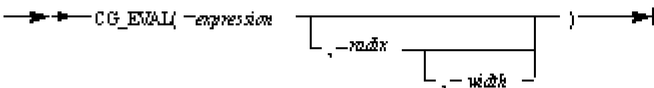
More complex mathematical operations are handled by CG_EVAL. This macro statement takes any *expression* and replaces it with the result. A *radix* can be applied to work in bases other than 10 (the *radix* must be from 2 to 36 inclusive). A *width* can be applied to make the result be padded with 0 (zero) to at least the number of characters specified by the *width* parameter. If the *width* is less than the length of the result, (numbers count) then no truncation occurs.

Syntax

Macro CG_EVAL Syntax

cg_eval_statement:

CG_EVAL '(' *expression* [, *radix* [, *width*]] ')'



Where:

- *expression* can contain various operators, as shown in the following table in decreasing order of precedence:

Operators Used in Expressions

Operator	Definition
-	Unary minus
**	Exponentiation
* / %	Multiplication, division and modulo

+ -	Addition and subtraction
<< >>	Shift left or right
== != > >= < <=	Relational operators
!	Logical negation
~	Bitwise negation
&	Bitwise and
^	Bitwise exclusive-or
/	Bitwise or
&&	Logical and
//	Logical or

All operators, except exponentiation, are left associative.
With relational operations, the value 1 is returned if it evaluates to True, otherwise the value is 0.

Example: Using CG_EVAL

*CG_EVAL(-2 * 5)* results in -10.
CG_EVAL(CG_INDEX(Good day, oo) > 0) results in 1.

The following statements result in 64:

```
CG_DEFINE(cube, <:CG_EVAL(($1)3):>)
cube(4)
```

Platform Support and Target Language Specifics

AppBuilder 3.2 Rules Language Reference Guide

Using platform-specific Rules Language in AppBuilder, it is possible to translate a Rules Language application to C, Java, ClassicCOBOL, and OpenCOBOL. Most of the Rules Language elements are supported for every target language, but there might be differences in syntax, semantics, and applicable functions. Complete descriptions of the Rules Language elements for each language can be found in the following sections:

- Specific Considerations for C
- Specific Considerations for Java
- Specific Considerations for CSharp
- Specific Considerations for ClassicCOBOL
- Specific Considerations for OpenCOBOL
- Specific Considerations for ClassicCOBOL and OpenCOBOL
- Restrictions on Features
- Supported Functions by Release and Target Language

Specific Considerations for C

Specific Considerations for C

The following sections describe specific differences in Rules Language elements for C:

- Data Types in C
- Comparing Views in C
- Object Method Call in C
- Date and Time Functions in C
- Common Procedures in C
- Constructing an Event Handler in C
- OVERLAY Statements in C
- Subscript Control in C

Restrictions on features are summarized in Restrictions on Features. To see which functions are supported, refer to Supported Functions by Release and Target Language.

Data Types in C

This section contains special considerations for using data types in C. For information about data types, refer to Data Types.

Alias in C

In C development, you can only declare object aliases for the window objects. See Alias for information about the Aliases object data item.

Variable for the Length of the VARCHAR Data Item in C

Changing the _LEN variable only affects the *LEN variable, the corresponding VARCHAR is _not* affected immediately. Therefore, any value is allowed for a _LEN variable, for example:

```
MAP -1 TO VC_LEN
MAP VC_LEN TO SomeVariable
```

SomeVariable will contain -1.

However, changing the _LEN variable affects how the string is interpreted in comparisons and in constructions, for example:

```
MAP "some string" TO VC
MAP 0 TO VC_LEN
IF VC = ""
  TRACE("VC is empty")
ENDIF
```

The trace statement will be executed because the length of the *VC* variable is set to zero, and *VC* becomes "(empty string).
Do not use the _LEN variable for write access (modifying the VARCHAR variable through its _LEN variable) in C.

Comparing Views in C

Comparison is a byte-by-byte memory comparison. If two views being compared are of unequal lengths, the shorter view is padded with blanks to equal the length of the longer view before the comparison is performed.
Because view comparison does not take into account the data type of the fields in the view, it is possible for the comparison of two views to give a different result than the comparison of the fields in the view.
For more information, refer to the following section: Comparing Views.

Object Method Call in C

In C, Method can have optional parameters, which can be omitted when Method is called. For example: If A is the object of class CLS, B is the method of class CLS with three parameters where the second parameter is optional and C is the method of class CLS, which has three parameters where the third parameter is optional, then the methods are used in the following way:

```
A.B (D,, E)
A.C (F,G,)
```

Example: Setting parameters for a C application

```
DCL
b object 'EXIT_BUTTON';
i smallint;
ENDDCL

map b.IsEnabled to i   *> This... <*
map b.IsEnabled() to i *> ...and this call are equivalent <*
```

Date and Time Functions in C

If you omit the format string, the following default format string will be provided:

- The format string that is specified by the DFLTDTFMT or DFLTTMFMT setting of the [CODEGENPARAMETERS] section of HPS.INI.
- If HPS.INI settings are not specified, ISO formats are used: "%Y-%0m-%0d" for DATE and "%0t.%0m.%0s" for TIME.

For more information, see Date and Time Function Definitions.

Common Procedures in C

For C, you can use OBJECT POINTER, but OBJECT ARRAY and OBJECT *cannot* be used as procedure parameters.
For more information, refer to Common Procedure Syntax and Procedure Syntax.

Constructing an Event Handler in C

The following constructions *cannot* be used in the event handler body in C:

- LOCAL PROCEDURE CALL
- USE RULE
- USE COMPONENT
- CONVERSE WINDOW
- RETURN

OVERLAY Statements in C

The OVERLAY statement in AppBuilder performs a byte-by-byte memory copy. Because the OVERLAY statement bypasses the MAP statement safety mechanism, it can cause unexpected results. The MAP statement carefully compares view structures to make sure that each data is mapped only into a field of the compatible data type; however, the OVERLAY statement blindly copies all the source data in its stored form to the destination data item. Erroneous OVERLAY statements might not be noticed during compilation but can result in problems during execution.
Refer to OVERLAY Statement for more information about the OVERLAY statement.

Data items of types MIXED and DBCS cannot be used in OVERLAY statements.

Applications that contain OVERLAY statements with data types that are not explicitly mentioned in this book are vulnerable to future failure. There is no guarantee that such applications can be ported to other platforms or supported from release to release. Therefore, as a precautionary measure, use MAP statements in all cases where OVERLAY statements are not necessary.

Subscript Control in C

Subscript control of occurring views performed in C relies on two Hps.ini settings: INDEX_CONTROL_ON and INDEX_CONTROL_ABORT; both can have YES or NO values.
If the INDEX_CONTROL_ON is set to YES (the default value), the view subscript control code is generated and application behavior is controlled by the INDEX_CONTROL_ABORT setting. If the INDEX_CONTROL_ABORT is set to YES, the application aborts when a view subscript is out of range. The default value is NO, and the default behavior is that if a "subscript is out of range" error occurs, the first occurrence is assumed, and the application continues to execute.
If the INDEX_CONTROL_ON is set to NO, the application does not abort and the first occurrence is assumed if the subscript is out of range without any notification, and the value of the HPSError is set to the corresponding error code. For detailed information about error messages, see the *Messages Reference Guide* . The INDEX_CONTROL_ABORT does not affect the application behavior.
See also -I code generation parameter.

Example: Subscript Control in C

In the following example, INDEX_CONTROL_ON is set to YES and INDEX_CONTROL_ABORT is set to NO:

```
DCL
I INTEGER;
V(10) VIEW CONTAINS I;
INDX INTEGER;
ENDDCL

MAP 10 TO INDX
MAP 1 TO I(INDX) *> Correct, I(10) is set to 1 <*
MAP -1 TO INDX
MAP 1 TO I(INDX) *> Error: index less than one, the first occurrence assumed <*
TRACE(I(1))      *> "1" is printed <*
IF HPSERROR = 6
   *> This line is executed and the message "ERROR : Index out of bounds" is printed <*
   TRACE("ERROR : Index out of bounds")
ENDIF
MAP 11 TO INDX
MAP 2 TO I(INDX) *> Error: index greater than view size, the first occurrence assumed <*
TRACE(I(1))      *> "2" is printed <*
RETURN
```

Specific Considerations for Java

The following sections describe the specific differences in Rules Language elements for Java:

- Data Types in Java
- Data Items in Java
- Comparing Views in Java
- Object Method Call in Java
- Creating a New Object Instance in Java
- ObjectSpeak Conversions in Java
- Functions in Java
- Dynamically-Set View Functions in Java
- Local Procedure Declaration in Java
- Event Procedure Declaration in Java
- Defining Views in Java
- Constructing an Event Handler in Java
- SQL ASIS Support in Java
- Transaction Support in Java
- Subscript Control in Java
- PRAGMA Statements in Java
- Static and Static Final Methods and Variables in Java
- Event Handler Statement in Java
- OVERLAY Statements in Java
- CASEOF in Java
- USE RULE ... DETACH OBJECT Statement in Java
- CONVERSE REPORT Statement in Java

Restrictions on features are summarized in Restrictions on Features. To see which functions are supported, refer to Supported Functions by Release and Target Language.

Data Types in Java

This section contains special considerations for using data types in Java. For information about data types, refer to Data Types.
Each Rules Language data type has its representation as one of the Java data types. This representation can be obtained from external Java classes, for example, from Java components using the getJavaValue method of `appbuilder.util.*` classes. The following data types are described in this section:

- ARRAY Object in Java
- INTEGER in Java
- SMALLINT in Java
- DEC and PIC in Java
- CHAR and VARCHAR in Java
- LONGINT, FLOAT and DOUBLE in Java
- Variable for the Length of the VARCHAR Data Item in Java
- DBCS and MIXED Data Types in Java
- DATE and TIME in Java
- TIMESTAMP in Java
- BOOLEAN in Java

- TEXT and IMAGE in Java
- OBJECT and OBJECT POINTER in Java

ARRAY Object in Java

Since ARRAY data items are only declared locally to a rule, and not in the hierarchy, they are not available to rules or components the declaring rule calls in the same way that views and fields are available. However, an ARRAY object can be passed to another rule or component using an OBJECT REFERENCE field in the view passed. The rule or component receiving the data must assign the OBJECT item to a locally declared ARRAY of the same type before accessing its contents.

INTEGER in Java

Java value has a type int. It is the value of the INTEGER variable.

SMALLINT in Java

Java value has a type short. It is the value of the SMALLINT variable.

DEC and PIC in Java

In the AppBuilder framework, DEC and PIC fields are stored as java.math.BigDecimal.

PIC with trailing sign

In Java, it is possible to declare a PIC data item with trailing sign, for example `p PIC'9V9S'`. By default, picture data item declared this way behaves like a usual signed picture (with leading sign): it occupies Length + 1 characters and can be used in any operation where picture with leading sign can be used with the same result.

However, picture with trailing sign is a separate data sub-type, and its internal character representation within the memory can be redefined based on a custom data converter.

AbfCOBOLDataConverter, the data converter different from the default one included in the AppBuilder java runtime package for emulating the COBOL applications data flow, processes the picture with trailing sign differently. With this converter, the picture with trailing sign occupies Length characters and is internally represented as a COBOL picture with sign trailing inclusive.

CHAR and VARCHAR in Java

Java value has a type `java.lang.String`. It is the value of the CHAR or VARCHAR variable.

LONGINT, FLOAT and DOUBLE in Java

Three additional data types are available for Java generation:

- LONGINT
- FLOAT
- DOUBLE.

LONGINT

Java value has a type long. The LONG data type is a 64-bit signed integer. It has a minimum value of -9,223,372,036,854,775,808 (-2^{63}) and a maximum value of 9,223,372,036,854,775,807 ($2^{63}-1$) (inclusive). Use this data type when you need a range of values wider than those provided by INTEGER.

FLOAT

Java value has a type float. The FLOAT data type is a single-precision 32-bit floating point number. Depending on the FLOATING_POINT_STANDARD hps.ini value, it can either be an IEEE 754 float or a float in HEXADECIMAL format. Its range of values is set between $-(2^{128}-2^{104})$ (approximately -3.4028234E38) and $2^{128}-2^{104}$ (approximately 3.4028234E38). Use a float (instead of double) if you need to save memory in large arrays of floating point numbers, or when single precision is enough. This data type should never be used for precise values, such as currency. For that, you can use the DEC type instead.

DOUBLE

Java value has a type double. The DOUBLE data type is a double-precision 64-bit floating point number. Depending on the FLOATING_POINT_STANDARD hps.ini value, it can either be an IEEE 754 double or a double in HEXADECIMAL format. Its range of values is set between $-(2^{1024}-2^{971})$ (approximately -1.79769E308) and $2^{1024}-2^{971}$ (approximately 1.79769E308). For decimal values, you should use a DEC type instead. As mentioned above, this data type should never be used for precise values, such as currency.

Depending on the FLOATING_POINT_STANDARD hps.ini value, the value ranges for FLOAT and DOUBLE types are different, as shown in Value ranges for FLOAT and DOUBLE:

Value ranges for FLOAT and DOUBLE

Type	Max (Type)	Min (Type)	Smallest positive value (unnormalized) SPV (Type)	Maximum non-zero digits to display MD (Type)	Number of precise digits (in mantissa) PD (Type)
IEEE 754 STANDARD					
FLOAT	$2^{128}-2^{104}$ (approximately 3.4028234E38)	$-(2^{128}-2^{104})$ (approximately -3.4028234E38)	2^{-149} (approximately 1.4012E-45)	8	7
DOUBLE	$2^{1024}-2^{971}$ (approximately 1.79769E308)	$-(2^{1024}-2^{971})$ (approximately -1.79769E308)	2^{-1074} (approximately 4.94E-324)	17	16
HEXADECIMAL FORMAT					
FLOAT	$16^{63}-16^{57}$ (approximately 7.2370E+75)	$-(16^{63}-16^{57})$ (approximately -7.2370E+75)	2^{-149} (approximately 1.4012E-45)	6	6
DOUBLE	$16^{63}-16^{57}$ (approximately 7.2370E75)	$-16^{63}-16^{57}$ (approximately -7.2370E75)	2^{-1074} (approximately 4.94E-324)	17	15

Three special values are supported for FLOAT and DOUBLE data items – positive infinity, negative infinity, and Not-a-Number (NaN) value, as described below:

- A *positive infinity* is assigned to a data item if there is an attempt to assign it a value exceeding Max(type).
- A *negative infinity* is assigned to a data item if there is an attempt to assign it a value less then Min(type).
- A *NaN* is assigned to a data item if there is an attempt to assign it a result of incorrect operation (i.e. 0/0).

A value, HEXADECIMAL of FLOATING_POINT_STANDARD hps.ini, is designed for emulating COBOL COMP-1 and COMP-2 data types.

> *Always use IEEE754 when you do not need to emulate the behavior of COBOL types!*

Variable for the Length of the VARCHAR Data Item in Java

Changing the `LEN` variable immediately changes the corresponding VARCHAR data. If `LEN` is assigned a negative value, zero length is assumed. If `LEN` is assigned more than the VARCHAR maximum length, the maximum length is assumed.

For more information refer to the following section: Variable for the Length of the VARCHAR Data Item.

Example: Using `LEN` variable in Java

Example 1
In the following example, because a negative value is assigned to `VC_LEN` , the value of `VC_LEN` becomes 0.

```
MAP -1 TO VC_LEN
TRACE(VC_LEN) // 0 will be printed
```

Example 2
You can safely modify the `LEN` field of VARCHAR without restrictions as shown in the following example.

```
DCL
  VC1 VARCHAR(10);
  VC2 VARCHAR(20);
ENDDCL

MAP "12345" TO VC1
MAP 10 TO VC1_LEN
MAP VC1 TO VC2
MAP VC2 ++ "A" TO VC2 // VC2 will contain '12345     A' (five spaces before A).
```

DBCS and MIXED Data Types in Java

Java value has a type `java.lang.String`. It is the value of the DBCS or MIXED variable. Trailing blanks are trimmed. DBCS characters are converted to Unicode (`java.lang.String`).

Because of the differences in character representation on different platforms, a varied number of characters can fit into a particular MIXED field. Keep the following in mind when writing Java applications:

- The length of a MIXED data item is calculated in characters in Java and can have a maximum length of 32K.
- In Java, each character (whether double- or single-byte) occupies one position in a MIXED string

For more information about the DBCS and MIXED data types refer to DBCS and MIXED Data Types.

DATE and TIME in Java

Java value has a type `java.util.Date`. It is 00.00.00 of the date value in the DATE variable in the local time zone. Java values of the same DATE variables are different in different time zones.

The Java value type `java.util.Date` is the time value in the TIME variable at January 1st, 1970 (Java "epoch" date) in the local time zone. Java values of the same TIME variable are different in different time zones.

For example: Washington, DC, USA is in the GMT -05:00 time zone. St. Petersburg, Russia is in the GMT +04:00 time zone. The Java value of the DATE variable representing June 03, 1999 is `java.lang.Date`, which corresponds to June 03, 1999 04:00:00 GMT on the computer running in St. Petersburg and June 02, 1999 19:00:00 GMT on the computer running in Washington.

TIMESTAMP in Java

Java value has a type `java.util.Date`. It is the moment of time contained in the TIMESTAMP variable in local time.

BOOLEAN in Java

Java value has a type boolean. It is the value of the BOOLEAN variable.

TEXT and IMAGE in Java

Java value has a type `java.lang.String`. It is the value of the TEXT or IMAGE field (file name).

OBJECT and OBJECT POINTER in Java

Java value has a type `java.lang.Object`. It is an object referenced by the OBJECT variable.

In Java, the OBJECT data type is equivalent to the OBJECT POINTER data type. This data type represents a non-typed reference to an object. Since any object of any class could be mapped to the OBJECT data type, it is useful when you want to perform a type conversion.

For more information about the OBJECT data type refer to OBJECT.

Using Object Data Types in Java

The declaration of OBJECT TYPE is equivalent to the OBJECT POINTER declaration. New objects created using OBJECT TYPE can only be used in Java application development.

Example: Using OBJECT in Java

The following are examples of different ways to use the OBJECT data type in Java.

Example 1: Using OBJECT data type in Java

Because the OBJECT data item and the OBJECT POINTER data item are treated the same way in Java, either one of them can be mapped to the data item of the OBJECT type.

```
DCL
  obj OBJECT;
  radio OBJECT TYPE RadioButton OF GUI_KERNEL;
  push OBJECT POINTER TO PushButton OF GUI_KERNEL;
ENDDCL

MAP radio TO obj
MAP push TO obj
```

Example 2: Using OBJECT for conversion in Java

In the following example, by making the object `radio` as the OBJECT data type, a procedure can be applied.

```
DCL
  obj OBJECT;
  radio OBJECT TYPE RadioButton OF GUI_KERNEL;
  resizeComponent PROC (comp OBJECT TYPE 'javax.swing.JComponent';
ENDDCL

resizeComponent(radio) *>Illegal: type of object "radio" is
                         incompatible with type of procedure formal
                         parameter <*

MAP radio TO obj

resizeComponent(obj) *>Valid: since obj has type OBJECT and this type
                       represents non-typed reference<*
```

Example 3: OBJECT declaration

The following example declares objects of type `java.awt.Button` and a local procedure with a parameter of the same type.

```
DCL
  java_button1 OBJECT TYPE 'java.awt.Button';
  java_button2 OBJECT TYPE 'java.awt.Button' OF JAVABEANS;
  button_proc PROC ( btn OBJECT TYPE 'java.awt.Button' );
ENDDCL
```

Using OBJECT POINTER in Java

The OBJECT POINTER data type is equivalent to the OBJECT data type. OBJECT POINTER TO represents a reference to an object of particular type. Use OBJECT POINTER TO to declare a pointer to an object.

The OBJECT POINTER is initialized with a NULL value. Use a MAP statement to assign a value to an object of the OBJECT OINTER data type.

The OBJECT POINTER data type is still supported in Rules Language for backward compatibility with AppBuilder 5.4.0. Do not use the OJBECT POINTER data type for new applications development.

Example: Using Object Pointer

Example 1: Object Pointer Declaration

In the following example, `push1` is the system identifier (HPS ID) or alias of a push button on a window that the rule converses. The method names and the types in this example correspond to a C Language application.

```
DCL
  mybutton OBJECT POINTER TO PushButton;
ENDDCL

MAP push1 TO mybutton
```

Example 2: Object Pointer as Parameter

An object pointer is particularly useful as a parameter to a common procedure. By declaring a pointer as a parameter, the procedure deals with any object of a particular type. For example, the following procedure enables an edit field, makes it visible, and sets its foreground color. To invoke the procedure, pass the name of a particular edit field.

```
PROC enableField (myField OBJECT POINTER TO EditField)
  MyField.Enabled(1)
  myField.Visible(1)
  myField.ForeColor( RGB(175,200,90) )
ENDPROC
.
.
enableField( field01 )
```

Example 3: Object Pointer in Event Procedure

Events often include parameters. Use an object pointer in an event procedure (see Event Handling Procedure) to represent a parameter of type POINTER or OBJECT received from an event triggered by a control. For example, the following procedure handles Initialize events from a rule window. In this example, the parameter passed by the window is a pointer to an object of type InitializeEvent.

```
PROC InitWindow FOR Initialize OBJECT MY_WINDOW
   (p OBJECT TYPE InitializeEvent)
ENDPROC
```

In this procedure:

- `InitWindow` is the procedure name.
- `Initialize` is the type of event handled.
- `MY_WINDOW` is the system identifier of the rule's window.
- `p` is the name (in the procedure) of the parameter received with the Initialize event from `MY_WINDOW`.
- `InitializeEvent` is the type of object to which a parameter points.

Assigning Object Data Type Variables in Java

In Java, variables of type OBJECT hold references to object instances. When a data item is assigned to a variable, the reference to the existing object is also assigned. To create new instances of an object, use the NEW clause. (See Creating a New Object Instance in Java for additional information.)

Example: Assigning References to Objects

```
DCL
  button1, button2 object type 'appbuilder.gui.AbfPushButton';
  ExitButton object 'EXIT'; *> Let "EXIT" be ID of EXIT button <*
  name char(100);
ENDDCL

PROC assignExample
  MAP ExitButton to button1 *>Now button1 holds ref to EXIT button<*
  MAP button1 to button2 *>and button2 too <*
  MAP button2.text to name *> name equals "EXIT" <*
  button2.setText('QUIT')
  MAP ExitButton.text to name *> name equals "QUIT" <*
ENDPROC
```

Implicit Numeric Conversions in Java

In Java, values of type FLOAT are implicitly converted to values of type INTEGER by dropping decimal part. This implicit conversion may occur, for example, in MAP statement or during passing of parameters.

Example: Implicit numeric conversions in Java

```
DCL
  f FLOAT;
  i INTEGER;
ENDDCL

PROC p(i1 INTEGER)
  TRACE(i1)
ENDPROC

MAP 1.1 to f
MAP f to i
TRACE(i)
p(f)

MAP 1.9 to f
MAP f to i
TRACE(i)
p(f)
```

In this example the output will be:

```
0 INFO  [APP]  1
0 INFO  [APP]  1
0 INFO  [APP]  1
0 INFO  [APP]  1
```

Data Items in Java

See the following for specific considerations when using data items in Java:

- Initialization in Java
- NULL in Java
- Default Object in Java

Initialization in Java

All variables are initialized with a NULL value in Java (see NULL in Java description). However, if a variable with a NULL value is used in a Rules Language expression where its particular value is required, an initial value that corresponds to the variable type is assumed. For example, if a BOOLEAN variable that has a NULL value is used as an IF condition, a FALSE value is assumed.

The CLEAR function resets a variable value to its initial value (see CLEAR Statement). This function sets a variable value to NULL in Java just as

the internal initialization routines does.

NULL in Java

A NULL value indicates no value. In Java development, variables of all data types can have NULL values. In Java, all variables are initialized with a NULL value. If a variable with a NULL value is used in a Rules Language expression where its particular value is required, an initial value corresponding to variable type is assumed. For example, if NULL BOOLEAN variable is used as an IF condition, a FALSE value is assumed.

The ISCLEAR Operator, CLEARNULL in Java, and ISNULL in Java functions manage the NULL attribute.
For more information, refer to Initializing Variables.

Default Object in Java

In Java, the following variables can be accessed in a rule without declaring them in the declaration (DCL) section.

- ***A Window variable of the type OBJECT TYPE***
  This variable is initialized to an instance of a window conversed by the rule.
  Address this variable in the rule as: *Variable* < *Window long name* >
- ***A Rule variable of the type OBJECT TYPE***
  This variable is initialized to an instance of the executing rule.
  When this variable is addressed in the rule as *Variable* < *Rule long name* >, if the Window long name is the same as the Rule long name, this variable is not created.
  When this variable is addressed in the rule as *Variable ThisRule* , it can still be used even if the Window long name is the same as the Rule long name.
- ***A Set variable of the type OBJECT TYPE***
  This variable is initialized to an instance of a Set with the long name. See *ObjectSpeak Reference Guide* for a description of Set object and dynamic set behavior.
  Address this variable in the rule as: *Variable* < *Set long name* >
- ***Variables that have the same names as the system identifiers of the Window objects.***
  The variables must have been initialized according to their types. If the system identifier is not a valid Rules Language identifier, a variable for it is *not* created. This system identifier can still be used in a rule by creating an alias for it (variable of type OBJECT 'HPSID').

See Data Items and Using Entities with Equal Names for more details on the naming restrictions.

Comparing Views in Java

Views with the same structure, meaning that the number, order and names of fields coincide in views being compared, are compared field by field, recursively. Views with different structures are compared as overlayed values; each view is overlayed to a string, then the strings are compared and return the result as the result of the views compare. See also Comparing Character Values for more information.

See also Comparing Views for more information.

Object Method Call in Java

The following is an example of setting parameters for a Java application.

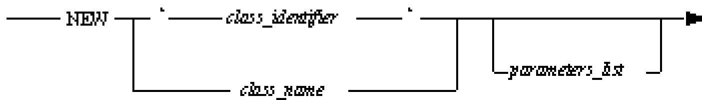
Example: Setting parameters for a Java application

```
DCL
  b object 'EXIT_BUTTON' ;
  Str VARCHAR(50);
ENDDCL

PROC Button1Click For Click OBJECT b
  (p OBJECT TYPE ClickEvent)
  map b.Text to Str *> This... <*
  map b.Text() to Str *> ...and this call is equivalent <*
ENDPROC
```

Creating a New Object Instance in Java

This clause is used in the Java application development to create new instances of objects.

NEW Syntax



where:

- *parameters_list* is the list of object constructor parameters included in round brackets; if constructor has no parameters then empty brackets must be omitted.

The following is an example of creating a new object instance in Java.

Example: Creating a new object instance

```
DCL
  label VARCHAR(200);
  p OBJECT TYPE 'java.awt.Button';
ENDDCL

PROC CreateButton : LIKE p
  PROC RETURN (NEW 'java.awt.Button')
ENDPROC

PROC GetLabel(btn LIKE p) : VARCHAR(200)
  PROC RETURN (btn.getLabel())
ENDPROC

MAP CreateButton TO p
MAP NEW 'java.awt.Button'('label') TO p
MAP GetLabel(NEW 'java.awt.Button'('label')) TO label
```

ObjectSpeak Conversions in Java

This topic describes conversions performed between the Java standard data types and the Rules Language data types when passing parameters to and accepting return values from Java methods.

Numeric Type

Any Java value of type *char, byte, short, int, long, float* , or *double* can be converted to the value of any of the following types: SMALLINT, INTEGER, DEC, or PIC, and similarly, the types SMALLINT, INTEGER, DEC, or PIC can be converted to any Java value of type *char, byte, short, int, long, float* , or *double.*

For SMALLINT and INTEGER, conversion is straight forward. If the value is too large, it is truncated.
When converting from DEC or PIC to INTEGER, the fraction part is truncated. If the integer part does not fit into the integer type, the assigned value is unpredictable. The overflowed value is converted to zero (0).

When converting from INTEGER to DEC or PIC, if the integer value does not fit into the integer part, the overflowed value is truncated.
When converting from DEC or PIC to *floats* , the nearest representable value is used. For example, 0.1 cannot be represented exactly in *float* or *double* type.

It is possible to use NIL as an ObjectSpeak method call parameter of type OBJECT. NIL is generated as `null` in the resulting Java code. For more information about ObjectSpeak, refer to the *ObjectSpeak Reference Guide* .

String Type

Java value of type `java.lang.String` can be converted to the value of type CHAR, VARCHAR, DBCS, or MIXED, and similarly, the value of type CHAR, VARCHAR, DBCS, or MIXED can be converted to a Java value of type `java.lang.String`.

OBJECT Type

In Rules Language, all classes in Java and the OBJECT data types are mutually convertible to Java subclassing rules.

BOOLEAN Type

Java values of boolean type can be converted to type BOOLEAN, and type BOOLEAN can be converted to Java values.

Date and Time Type

Java values of type `java.util.Date` can be converted to types DATE, TIME, and TIMESTAMP, and types DATE, TIME, and TIMESTAMP can be converted to Java values of type fava.util.Date. Rules of conversion are the same as described in Data Types in Java.

Functions in Java

The following functions have specific considerations for Java:

- CHAR in Java
- CLEARNULL in Java
- Date and Time Functions in Java
- Double-Byte Character Set Functions in Java
- GET_ROLLBACK_ONLY in Java
- INCR and DECR in Java
- ISNULL in Java
- LOC in Java
- Format String Specific for FLOAT and DOUBLE Data Items
- Numeric Conversion Functions in Java
- RTRIM in Java
- SET_ROLLBACK_ONLY in Java
- STRLEN in Java
- SUBSTR in Java
- TRACE in Java
- UPPER and LOWER in Java
- VERIFY in Java

CHAR in Java

If the CHAR function is applied to an uninitialized numeric variable, meaning a variable that was initialized with a NULL value and never changed, then the value returned depends on the SHOW_ZERO_ON_NULL setting in the appbuilder.ini file. If this setting is TRUE, then the CHAR function will return a string containing the zero symbol; otherwise, an empty string is returned. For more information see CHAR.

The CHAR function supports LONGINT and floating point numbers only in Java and in OpenCOBOL according to the following syntax:

```
CHAR (LONGINT data item [, format string])
CHAR (FLOAT data item[, format string])
CHAR (DOUBLE data item[, format string]).
```

LONGINT numbers are formatted by the same rules as integers.

If no format string is provided for CHAR function with one parameter of type LONGINT, FLOAT or DOUBLE, then the default format string is used as the second parameter and the result is the same as for the CHAR function with two parameters.

Default format strings for LONGINT, FLOAT, and DOUBLE

Type	Default format string for CHAR function
LONGINT	"s9"
FLOAT	"SZZZZZZZZV9ZZZZZZZE%esZZZ%s"
DOUBLE	"sZZZZZZZZZZZZZZZZZV9ZZZZZZZZZZZZZZZZE%esZZZ%s"

For details about the floating point numbers, see Format String Specific for FLOAT and DOUBLE Data Items.

CLEARNULL in Java

This is available for Java only.

The CLEARNULL function takes a field or a view as an argument and clears the NULL flag of the field or every field in a view if it is applied to a view without changing the value of the field. After this function invocation, the field value is not changed and is no longer considered NULL.

The CLEARNULL support function *cannot* be applied to variables of any object type.

Example: Using CLEARNULL Function

In the following example, after the CLEARNULL function is applied to I, the NULL flag of I is cleared, therefore is no longer considered as NULL; however, I still contains its initial value.

```
DCL
  I INTEGER;
  B BOOLEAN;
ENDDCL

CLEARNULL(I) *>I is set to its initial value - zero<*
MAP ISNULL(I) TO B *>B is FALSE<*
MAP ISCLEAR(I) TO B *>B is TRUE, because I contains its initial value<*
RETURN
```

SIZEOF in Java

The implementation of a rule data converter enhancement in AppBuilder 3.2 makes it possible to configure SIZEOF behavior.

The default data converter is `appbuilder.util.AbfDefaultDataConverter`. This converter should be used if there is no reason to change the data conversion algorithm that is used for OVERLAY and REDEFINE operations.

The size of data item in java generation is counted by a data converter class. This class implements the `appbuilder.util.AbfDataConverter` interface, and its fully-qualified name should be specified in the DATA_CONVERTER hps.ini setting.

Date and Time Functions in Java

If you omit the format string, the following default format string is provided:

- Format string specified by the DEFAULT_DATE_FORMAT or the DEFAULT_TIME_FORMAT settings of the [NC] section of the appbuilder.ini file.
- If the appbuilder.ini setting is not specified, the default system value (Java regional setting) is used for Date.
- If is appbuilder.ini setting is not specified, then the parameter is considered to be the correct value of the TIME data type and is used as is, without any conversion.

For more information, see Date and Time Function Definitions.

Double-Byte Character Set Functions in Java

In Java, codepage validation is specified by the DBCS_VALIDATION_CODEPAGE parameter in the [VALIDATION] section of the appbuilder.ini file. This ini setting can be changed without recompilation. If validation fails, an exception is raised at runtime. In Java, these conversion functions just change the types of their arguments and perform validation as explained in Validation and Implementation of Double-Byte Character Set.

GET_ROLLBACK_ONLY in Java

This is available for Java only.

The GET_ROLLBACK_ONLY function returns a BOOLEAN value, indicating whether or not the only possible outcome of the transaction associated with the current thread is to roll back the transaction (TRUE) or not (FALSE).

INCR and DECR in Java

The following example illustrates how INCR and DECR functions are used in a MAP statement.

```
MAP 0 to I
MAP INCR(I) + DECR(I) + 1 to J
```

As a result, `I` is set to 0 and `J` is set to 2.
Refer to INCR and DECR in OpenCOBOL to see how the result is different using the same MAP statement.

ISNULL in Java

This is available for Java only.

The ISNULL function takes a field as an argument and returns a BOOLEAN value indicating whether the field is NULL or not. If the field's value is NULL, ISNULL returns TRUE, otherwise it returns FALSE.

The ISNULL support function *cannot* be applied to variables of any object type. (See Object Data Types.)

If you wish to test a variable of any object type (Object Data Types) for null, use ISCLEAR. It returns TRUE if this reference actually refers to nothing. In other words, it contains a null value and returns FALSE if it refers to some object (non-NULL value).

A field contains NULL if it has never been modified by a user or if it has been reset programmatically by using the CLEAR statement. This is not the same as the field initial value. For example, if you assign a value of 0 to an integer field, it is not NULL any longer (that is, ISNULL returns FALSE); however, ISCLEAR applied to this field returns TRUE as if the field has not changed.

Example: ISNULL, NULL, and cleared fields

Example 1 illustrates the use of ISNULL function, Example 2 illustrates the differences between NULL and cleared fields:

Example 1: Using ISNULL Function

```
DCL
  CH CHAR;
  I INTEGER;
  B BOOLEAN;
  OBJ OBJECT TYPE Rule;
  V VIEW CONTAINS CH;
ENDDCL

MAP ISNULL(I + 1) TO B
*>Compile time error: ISNULL cannot be applied to expression<*

MAP ISNULL(OBJ) TO B
*>Compile time error: ISNULL cannot be applied to object<*

MAP ISNULL(V) TO B
*>Compile time error: ISNULL cannot be applied to view<*

MAP ISNULL(CH) TO B
*>Since all fields upon rule start are initialized with NULL value,
B is TRUE <*

MAP ISNULL(I) TO B *>B is TRUE <*
MAP ISNULL(B) TO B *>B is FALSE, since B was assigned TRUE <*
RETURN
```

Example 2: Using Null and Cleared Fields

The following example illustrates the differences between null and cleared fields:

```
DCL
  I INTEGER;
  B BOOLEAN;
ENDDCL

MAP ISNULL(I) TO B *>B is TRUE<*
MAP ISCLEAR(I) TO B *>B is TRUE<*
MAP 0 TO I *>I contains initial value, that is, zero<*
MAP ISNULL(I) TO B *>B is FALSE, because I was modified<*
MAP ISCLEAR(I) TO B *>B is TRUE, because I contains
its initial value<*
RETURN
```

Format String Specific for FLOAT and DOUBLE Data Items

There are two different ways of displaying floating point numbers:

- Exponential Notation
- Non-exponential Notation

Exponential Notation

The FLOAT or DOUBLE format string consists of the formatted representation of two numbers – *mantissa* and an *integer exponent* (the latter is mandatory).
The format string must provide enough information to determine the mantissa and the integer exponent without ambiguity. You can obtain the following information from the format string and the environment:

- *Dint* = Dint(format string) – the maximum number of digits allowed by the format string in the integer part of the mantissa.
- *Dscale* = Dscale(format string) – the maximum number of digits allowed by the format string in the decimal part, which is also called *scale* , of the mantissa.
- *Representation mode* – the desired manner of a floating point number representation. There are two basic representation modes and an additional numeric representation mode:
- ***Standard*** – the string representation contains as many significant digits of the FLOAT/DOUBLE number as possible with given Dint and Dscale; if there are several such representations, then the one with the minimal absolute value of exponent(DOUBLE/FLOAT) is used.
- ***COBOL*** – mantissa(DOUBLE/FLOAT) is as large as possible with given Dint and Dscale. The following examples illustrate basic representation modes:
  Standard Representation Mode Examples

Parameters FLOAT/DOUBLE	**Dint=4, Dscale=1** **-ZZZ9V9E-Z9**	**Dint=5, Dscale=2** **-ZZZZ9V99E-Z9**	**Dint=1, Dscale=5** **-9V99999E-ZZ**
12.345	1234.5 E-2	123.45 E-1	1.23450E1
0.05	0.5 E-1	0.05 E0	0.05000
1	1.0 E0	1.00 E0	1.00000

COBOL Representation Mode Examples

Parameters FLOAT/DOUBLE	**Dint=4, Dscale=1** **-9999V9E-9**	**Dint=5, Dscale=2** **-99999V99E-9**	**Dint=1, Dscale=5** **-9V99999E-ZZ**
12.345	1234.5E-2	12345.00E-3	1.23450E1
0.05	5000.0E-5	50000.00E-6	5.00000E-2
1	1000.0E-3	10000.00E-4	1.00000

- ***Numeric representation mode*** is managed by a positive integer, Dexp:
  Dexp = Dexp(format string) – a positive integer in which the following applies:
- If 10^{1-Dexp} =< Abs(DOUBLE/FLOAT) < 10^{Dexp} , then the exponent is zero and the number is completely represented by its mantissa, if the given Dint and Dscale allows this representation.
  The meaning of *Dexp* parameter can be illustrated with the following examples:
  Here Dint is equal to Dscale, and both are large numbers exceeding the length of FLOAT/DOUBLE.
  Numeric Representation Mode Examples

Dexp FLOAT/DOUBLE	**1**	**2**	**3**	**4**
12.345	1.2345 E1	12.345 E0	12.345 E0	12.345 E0
123.45	1.2345 E2	12.345 E1	123.45 E0	123.45 E0
1234.5	1.2345 E3	12.345 E2	123.45 E1	1234.5 E0
0.12345	1.2345 E-1	0.12345 E0	0.12345 E0	0.12345 E0
0.012345	1.2345 E-2	0.12345 E-1	0.012345 E0	0.012345 E0
0.0012345	1.2345 E-3	0.12345 E-2	0.012345 E-1	0.0012345 E0

The table cells contain string representations of DOUBLE/FLOAT for different Dexp.
The format string for exponential notation consists of two parts: *mantissa* and *exponent* .
Mantissa can be formatted as decimal number using the existing set of format symbols.
The formatting function computes Dint and Dscale from the format string the following way:

- *Dint* is a number of digit symbols (9, Z, z, *) before the decimal separator (V, v) or a number of all digit symbols for mantissa, if there is no

decimal separator.

- *Dscale* is a number of digit symbols (9, Z, z, *) after the decimal separator (V, v) or 0 if there is no decimal separator. Dint = Dscale = 0 if no mantissa formatted.
  Exponent should be formatted as integer.
  The following symbols denote optional locale-specific exponent symbol and a representation mode (please notice that these symbols apply for DOUBLE and FLOAT only, and not for LONGINT!):
- ***%e*** – for the exponent symbol
- ***%s*** – for the Standard mode
- ***%c*** – for the Cobol mode
- ***%n*** – for the Numeric mode
- ***E*** – for the separation of exponent from mantissa

Symbol ***E*** is mandatory when using exponential notation. Other new symbols are optional. The default representation mode is set to *Standard* and can be configured in the appbuilder.ini file. It is used when the representation mode is not specified in the format string explicitly. If the representation mode is explicitly specified in the format string, it overrides the default value.

Symbol ***%n*** , if present, must be followed by at least one digit. All digits between %n symbol and the first non-digit symbol are treated as Dexp value.

The format string for the exponential notation of the floating-point number can be written as follows:

```
mantissa E exponent
```

where:

- *mantissa* is a correct simple format string, i.e. the format string for non-exponential notation (may be empty).
- *exponent* is a correct simple format string with at least one digit symbol, containing no V, v and. (dot) format symbols and containing not more than one%e format symbol at any position before the first digit symbol.

Additionally, one and only one symbol from the group – ***%s*** , ***%c*** , ***%n*** – is allowed in place of the format string.

Restrictions:

> *Simple format string restrictions apply to the format strings, mantissa and exponent. For more information, see Format string validation.*

1. No E symbol can occur in *simple format strings* , *mantissa* and *exponent* .
2. Only one $ symbol is allowed for the whole format string.
3. No V, v and . (dot) symbols are allowed in the *exponent* format string.
4. No %e symbol can occur in the *mantissa* format string.

If, for the given format string, *all digit symbols* in the *exponent* are Z or z and an *exponent* contains the %e symbol, and if the value of a number DOUBLE/FLOAT being formatted is such that an integer exponent (DOUBLE/FLOAT) = 0, then no exponent symbol is present in the result.

5. Exponent is never truncated if it has three digits and the format string has only two positions; all three digits are printed anyway.

Non-exponential Notation

In non-exponential notation, floating point numbers are formatted by the same rules as decimals. The restriction is that a format string *cannot contain* E, %e, %s, %c, %n symbols.

LOC in Java

In Java, the LOC function returns an object representing a given data field. Every data field in a rule is represented in Java by an instance of `appbuilder.util.AbfDataObject` descendant. The LOC function returns Rules Language OBJECT data type represented by Java class `appbuilder.util.AbfDataObject` referencing a given field or view. In Java, the LOC function can accept not only views but also fields as arguments. Untyped OBJECT is returned.

For more information refer to LOC.

Example: Using LOC Function in Java

The LOC function can be used to pass references to data items in a rule to Java classes.

```
DCL
  I INTEGER;
  V VIEW CONTAINS I;
  O OBJECT;
  MyMap OBJECT TYPE 'java.util.HashMap';
  Key VARCHAR(20);
  Value VARCHAR(255);
ENDDCL

MAP LOC(I) TO O
MAP LOC(V) TO O
MAP NEW 'java.util.HashMap' TO MyMap

MyMap.put(Key, Value) *> Illegal, wrong parameter types <*
MyMap.put(LOC(Key), LOC(Value))
*> Legal parameter types are java.lang.Object<*
```

Numeric Conversion Functions in Java

The numeric conversion functions use locale-specific decimal, thousand and currency tokens. For example, INT("123$") will return 0 under the German locale since $ is not recognized as a currency symbol. Below are some special considerations for the following functions:

- LONG in Java
- FLOAT in Java
- DOUBLE in Java.

LONG in Java

The LONG function is available for Java and OpenCOBOL only. This function converts a string to LONGINT, its function is analogous to INT function, but it returns the value of the LONGINT type. For details about INT function, see Numeric Conversion Functions.

The format symbols – %e, %s, %n, E – are not allowed in format strings for the LONGINT function. See also Exponential Notation.

Syntax:

```
LONG (character data item)
LONG (character data item [, format string])
```

FLOAT in Java

The FLOAT function is available for Java and OpenCOBOL only. This function converts a string to FLOAT.

Syntax:

```
FLOAT (character data item)
FLOAT (character data item [, format string])
```

DOUBLE in Java

The DOUBLE function is available for Java and OpenCOBOL only. This function converts a string to DOUBLE.

Syntax:

```
DOUBLE (character data item)
DOUBLE (character data item [, format string])
```

The list of valid format strings for FLOAT and DOUBLE functions is the same as for CHAR function (see CHAR for details), with the following constraints:

- If the format string does not use exponential notation, then the restrictions are the same as for the INT and DECIMAL functions.
- If the format string uses exponential notation, then the restrictions are the same as for the INT and DECIMAL functions being applied to the mantissa and exponent substrings correspondingly.

RTRIM in Java

If the RTRIM function is applied to an invalid DBCS string, the function returns the same invalid string with trailing DBCS blanks removed. For more information see RTRIM.

SET_ROLLBACK_ONLY in Java

This is available for Java only. This function has no parameters.

The SET_ROLLBACK_ONLY function modifies the transaction associated with the current thread so that the only possible outcome of the transaction is to roll back the transaction.

STRLEN in Java

When the STRLEN function is applied to a DBCS string, the function parameter is not required to be a valid DBCS string. When STRLEN is applied to a MIXED string, the function parameter is not required to be a valid MIXED string. For more information see STRLEN.

SUBSTR in Java

The MIXED or DBCS parameters can contain invalid characters. For more information see SUBSTR.

TRACE in Java

The code for TRACE() statements might be generated even with Rule debug option not selected in Construction Workbench > Options > Preparation tab. With ALWAYS_GENERATE_USER_TRACE codegen parameter from hps.ini set to YES, user-coded TRACE statements are always generated in the target code and can be enabled or disabled through runtime settings. TRACE function is output only if the APP_LEVEL setting in the [TRACE] section of the appbuilder.ini file is set to 1 or greater.

For more information refer to TRACE.

UPPER and LOWER in Java

Characters are converted to upper and lower case according to the specified codepage. In Java, this codepage is the current system codepage. For more information see UPPER and LOWER.

VERIFY in Java

The MIXED or DBCS parameters can contain invalid characters. For more information see VERIFY.

Dynamically-Set View Functions in Java

In Java, you can check or change the number of occurrences in views dynamically while a rule executes using the following standard functions:

- OCCURS
- APPEND
- RESIZE
- DELETE
- INSERT
- REPLACE

OCCURS

The OCCURS function returns the number of occurrences of a given *view* . For non-occurring views, it returns 0.

OCCURS Syntax

```
──OCCURS──(──view──)──────────────────────────────────────►
```

where:

- *view* is any view.

Example: Using OCCURS Function

In the following example, the OCCURS function is used to get the number of occurrences of views `V1`, `V2` and `V3`.

```
DCL
  I, J INTEGER;
  COUNT INTEGER;
  V1 VIEW CONTAINS I;
  V2 VIEW CONTAINS J;
  V3 VIEW CONTAINS I,J;
  V VIEW CONTAINS V1(10), V2, V3(1);
ENDDCL

MAP OCCURS(V1) TO COUNT *> COUNT=10 <*
MAP OCCURS(V2) TO COUNT *> COUNT=0 : V2 is not an occurring view <*
MAP OCCURS(V3) TO COUNT *> COUNT=1 : V3 is an occurring view,
though with 1 occurrence only <*

MAP OCCURS(V1(1)) TO COUNT *> COUNT=0 : not an occurring view <*
```

APPEND

The APPEND function appends the *source_view* to the *target_view* . Views must be identical in structure, meaning that the number, order and names of fields in the source_view must be the same as in the target_view.

APPEND Syntax

```
—APPEND——(—target view —, —source view — ┬——————————————————————————————————┬— ) —►
                                          └ , — number_of_occurs_to_process ┘
```

where:

- *target_view* must be an occurring view.
- *source_view* is any view.
- *number of occurs to process* parameter specifies how many items are taken from the *source_view* .

If the *number of occurs to process* is greater than the size of *source_view* , all items from it are used and a warning is issued at runtime. If this parameter is less than zero, then zero is assumed, and no item from *source_view* is taken.

Example: Using APPEND Function

In the following example, both `APPEND(V1, V2)` and `APPEND(V1, V4)` statements are successful because `V1` is an occurring view and `V1`, `V2` and `V4` have the same structure. `APPEND(V1, W)` fails because `V1` and `W` have different structures. `APPEND(V3, V1)` also fails because `V3` is not an occurring view.

```
DCL
  I INTEGER;
  C CHAR(10);
  V1, V2, V3, V4 VIEW CONTAINS I, C;
  W VIEW CONTAINS I;
  V VIEW CONTAINS V1(10), V2(14), V3, V4(10);
ENDDCL

MAP 1 TO I OF V1(1)
MAP 2 TO I OF V2(1)
MAP 4 TO I OF V4(1)
APPEND(V1, V2)
MAP OCCURS(V1) TO COUNT
TRACE(COUNT) *> Outputs: "24" <*
TRACE(I OF V1(1), I OF V1(11)) *> Outputs: "1 2" <*
APPEND (V1, V4)
TRACE (I of V1(25)) *>Outputs: "4" <*
APPEND(V1, W) *> Illegal: views of different structure <*
APPEND(V3, V1) *> Illegal: V3 is not an occurring view <*
```

RESIZE

The RESIZE function shrinks or expands the given occurring view to a new size.

RESIZE Syntax

```
— RESIZE ——(—target_view—,—new_size ————————————————————— ) —►
                                      |                    |
                                      └—,— from_position ——┘
```

where:

- *target_view* is an occurring view.
- *new_size* specifies the new size of the *target_view* .
- *from_position* specifies the starting position to apply the *new_size* within the *target_view* .

By default, the RESIZE function is applied to as many occurrences as possible starting from the beginning of the view. If the third parameter, *from_position* is specified, it keeps as many occurrences as possible starting from the specified position. Occurrences between the first position and *from_position* are then lost. If *from_position* is greater than the total number of occurrences, all occurrences are lost. If *from_position* is less than or equal to zero, RESIZE behaves as if *from_position* is not given.

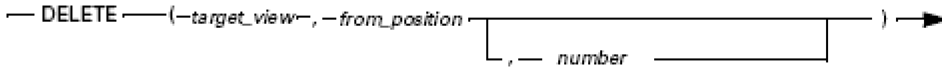
Example: Using RESIZE Function

```
DCL
  I, J INTEGER;
  V(10) VIEW CONTAINS I;
ENDDCL

MAP 27 TO I(1), I(2), I(3), I(10)
RESIZE(V, 20)
MAP I(10) TO J *> 27 <*
MAP I(15) TO J *> NULL - a new occurrence <*
RESIZE(V, 5)
MAP I(1) TO J *> 27 <*
MAP I(10) TO J *> runtime error -- too large subscript <*
RESIZE(V, 5, 2)
MAP I(1) TO J *> 27 <*
MAP I(3) TO J *> NULL -- 1st occurrence removed, so 3rd became 2nd <*
```

DELETE

The DELETE function deletes occurrences of a view starting from the position given in the second argument.

DELETE Syntax



where:

- *target_view* is an occurring view.
- *from_position* specifies the starting position to delete.
- *number* specifies the number of occurrences to delete.

By default, the DELETE function deletes occurrences of a view until the end of the view. If the third parameter, *number* is given, it deletes *number* occurrences starting from the given position. If there are not enough occurrences after the given position, it deletes as many as possible until the end of the view.

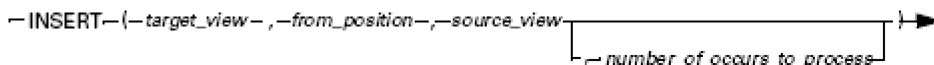
Example: Using DELETE Function

```
DCL
  I, J INTEGER;
  V(10) VIEW CONTAINS I;
ENDDCL

DELETE(V, 5) *> Deletes occurrences 5 through 10,occurrences 1 through 4 are still in the view <*
DELETE(V, 2, 1) *> Deletes occurrence 2, now there are only 3occurrences remaining <*
DELETE(V, 2, 10) *> Deletes only occurrences 2-3, only oneoccurrence is left in the view <*
```

INSERT

The INSERT function inserts all occurrences of the source view (or the view itself if it is the plain view) at the specified position in the target view. Views must be identical in structure, meaning that the number, order and names of fields in the source view must be the same as in the target view.

INSERT Syntax



where:

- *target_view* is an occurring view.
- *source_view* is any view.
- *from_position* specifies the position to insert.
- *number_of_occurs_to_process* specifies how many items are taken from *source_view* .

If the *number_of_occurs_to_process* is greater than the size of the *source_view* , all items from the *source_view* are inserted, and a warning is issued at runtime. If it is less than zero, then zero is assumed, and no item from *source_view* is taken.

Example: Using INSERT Function

```
DCL
  I INTEGER;
  C CHAR(10);
  V1, V2, V3 VIEW CONTAINS I, C;
  W VIEW CONTAINS I;
  V VIEW CONTAINS V1(10), V2(2), V3, W(10);
ENDDCL

MAP "O1V1" TO C OF V1(1)
MAP "O2V1" TO C OF V1(2)
MAP "O1V2" TO C OF V2(1)
MAP "O2V2" TO C OF V2(2)
MAP "O0V3" TO C OF V3
INSERT(V1, 2, V2)
TRACE(C OF V1(1)) *> O1V1 <*
TRACE(C OF V1(2)) *> O1V2 <*
TRACE(C OF V1(3)) *> O2V2 <*
TRACE(C OF V1(4)) *> O2V1 <*
INSERT (V1, 4, V3)
TRACE (C OF V1 (4)) *>O0V3 <*
INSERT(V1, 5, W) *> Illegal: W does not have identical structure as V1 <*
INSERT(V1, 27, V2) *> runtime error <*
```

REPLACE

The REPLACE function replaces occurrences of the *target_view* with occurrences from the *source_view* , starting from the specified position. Views must be identical in structure, meaning that the number, order and names of fields in the source_view must be the same as in the target_view.

REPLACE Syntax

```
—REPLACE—(—target_view— ,—from_position—,—source_view—┬────────────────────────────────┬—)—►
                                                       └—,—number_of_occurs_to_process—┘
```

where:

- *target_view* is an occurring view.
- *source_view* is any view.
- *from_position* specifies the starting position to replace.
- *number_of_occurs_to_process* specifies how many items are taken from *source_view* .

If the *from_position* is invalid (less than zero or greater than the size of the *target_view*), a runtime error is generated. Both the *source_view* and the *target_view* must be identical in structure. REPLACE does not add occurrences, so only those occurrences that exist in the target view are replaced.

If the *number_of_occurs_to_process* is greater than the size of the *source_view* , all items from it are used and a warning is issued at runtime. If it is less than zero, then zero is assumed, and no item from *source_view* is taken.

Example: Using REPLACE Function

```
DCL
  I INTEGER;
  C CHAR(10);
  V1, V2, V3 VIEW CONTAINS I, C;
  W VIEW CONTAINS I;
  V VIEW CONTAINS V1(10), V2(2), V3, W(10);
ENDDCL

MAP "O1V1" TO C OF V1(1)
MAP "O2V1" TO C OF V1(2)
MAP "O1V2" TO C OF V2(1)
MAP "O2V2" TO C OF V2(2)
MAP "O0V3" TO C OF V3
MAP "Test" TO C OF V1(4)
REPLACE(V1, 2, V2)
TRACE(C OF V1(1)) *> O1V1 <*
TRACE(C OF V1(2)) *> O1V2 <*
TRACE(C OF V1(3)) *> O2V2 <*
TRACE(C OF V1(4)) *> Test <*
REPLACE (V1, 4, V3)
TRACE (C OF V1(4)) *> O0V3 <*
REPLACE(V1, 5, W) *> Illegal: W does not have identical structure as V1 <*
REPLACE(V1, 27, V2) *> runtime error <*
REPLACE(V1, 10, V2) *> V1(10) is replaced with V2(1) <*
```

Local Procedure Declaration in Java

For Java, any data type can be used as a procedure parameter.

For more information about the local procedure declaration, refer to Local Procedure Declaration.

Event Procedure Declaration in Java

The following example illustrates the declaration of event procedures in Java. For more information refer to Event Procedure Syntax. For a detailed list of supported events, see the *ObjectSpeak Reference Guide* .

Example: Java LISTENER Clause

In this example, two handlers, `hand1` and `hand2`, are declared for the event keyPressed for objects of type `java.awt.Button`.

```
DCL
  hand1 proc for keyPressed type 'java.awt.Button'
      ( evt object 'java.awt.event.KeyEvent' );
  hand2 proc for keyTyped listener 'java.awt.event.KeyListener'
      type 'java.awt.Button' (evt object
          java.awt.event.KeyEvent' );
ENDDCL
```

Although the first definition has no LISTENER clause, it is equivalent to the second definition.

```
hand3 proc for keyPressed type 'java.awt.Button'
    ( evt object 'java.awt.event.KeyEvent' );
```

This handler declaration is equivalent to the previous two.

```
DCL
  event object 'java.awt.event.KeyEvent';
  button object type 'java.awt.Button';
  ButtonPtr object 'java.awt.Button';
  HandlerForPointer proc for keyPressed object ButtonPtr
      ( evt like event );
  HandlerForObject proc for keyPressed object button
      ( evt like event );
ENDDCL
```

The event handlers `HandlerForPointer` and `HandlerForObject` are declared for the distinct objects referenced respectively by the variables `ButtonPtr` and `Button`. In both cases, using the LISTENER clause produces the same result.

Defining Views in Java

In Java, you can define a local procedure that has one or more parameters of the view type without defining the view type. This procedure call can get any view as an actual parameter.

If a procedure has a non-typed view as a parameter, then this parameter is passed by reference, as opposed to typed parameters that are always passed by values in the Rules Language. Please notice that if you use this specific procedure declaration, you might encounter a severe reduction of runtime performance compared to the procedure with type parameters. This is the case when your procedure call can copy the views of different types with copying source or destination being coded as a non-typed view parameter. In this case, the reflection-based mapping algorithm is used and performs poorly compared to the direct views copying generated code when the views' types are known at compile-time.

A non-typed view can only be used in its entirety in a MAP statement, when passed as a parameter to another procedure or subrule call, or as an argument of CLEAR operator, CLEARNULL, LOC, OCCURS, and SIZEOF functions. In these cases, view mapping is performed dynamically during execution according to the view mapping algorithm. For a description of the view mapping algorithm, see Assignment Statements.

Example: View Mapping

```
DCL
  i, j integer;
  u view contains i;
  w view contains i, j;
  z view contains u, w;
ENDDCL

PROC p ( v view )
  MAP v to u
ENDPROC

p(w) *> Here i of w is assigned to i of u <*
p(z) *> No assignments is performed - i is on different level in z <*
p(u) *> i of u assigned to i of u <*
```

Constructing an Event Handler in Java

The following constructions *can* be used in the event handler body in Java:

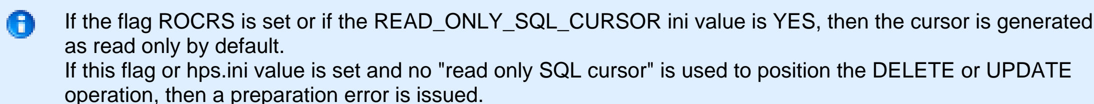
- Local procedure call
- USE RULE
- USE COMPONENT
- CONVERSE WINDOW
- RETURN

SQL ASIS Support in Java

Java does not support embedded SQL; it supports SQLJ. AppBuilder generates SQLJ from the code in SQL ASIS. The following restrictions apply:

- ***Dynamic SQL***
  Dynamic SQL is not supported. An error might be generated if a dynamic SQL statement is found in SQL ASIS code, but not all errors are reported. This allows more flexible DBMS support.

- ***NULL value***
Indicator variables only indicate that an associated host variable has a NULL value. No other errors are indicated. You can also use ISNULL to test for a NULL value in a field along with the PROPAGATE_NULL_TO_DATABASE=TRUE in the appbuilder.ini file. This is because NULL values are supported for Java.
- ***Cursor generation***
By default, a cursor is generated FOR UPDATE except in the following cases, which generate the cursor implicitly READ ONLY. The following fullselects and select-clauses refer to the DECLARE CURSOR statements only:
  - the outer fullselect includes a GROUP BY clause or HAVING clause
  - the outer fullselect includes column functions in the select list
  - the outer fullselect includes a UNION or UNION ALL clause
  - the select-clause of the outer fullselect includes a DISTINCT clause
  - the select-statement includes an ORDER BY clause
  - the select-statement includes a READ ONLY clause
  - the select-statement includes a FETCH FIRST *n* ROWS ONLY clause.



If the flag ROCRS is set or if the READ_ONLY_SQL_CURSOR ini value is YES, then the cursor is generated as read only by default.
If this flag or hps.ini value is set and no "read only SQL cursor" is used to position the DELETE or UPDATE operation, then a preparation error is issued.

- ***Persistent SQL cursor in Java***
The *Persistent SQL cursor in Java* refers to the possibility that a cursor created by a particular rule must be retained past the end of the rule invocation and made available to subsequent invocations of that rule within the same scope until explicitly closed; however, in the case of executing in a server request scope, the request terminates.
Depending on the rule/type execution environment, the persistent cursor scope can be defined as follows:
  - For *Server Rules* (EJB, Web Services, RMI)
  The scope is limited by the server request. A persistent cursor that is opened by a particular rule is available to that rule whenever the server performs the request until the time at which the cursor is closed.
  - For *Client Rules* (GUI, HTML) – non-detached
  The scope is limited to the non-detached rules within the client execution environment. A persistent cursor opened by a particular non-detached rule is available to non-detached instances of that rule whenever it is called within the client-side application until the cursor is closed. Detached instances of that rule do not have access to a persistent cursor opened by a non-detached instance.
  - For *Client Rules* (GUI, HTML) – detached
  The scope is limited to the detached instance. A persistent cursor opened by a particular rule within a detached instance is available to that rule within the detached instance until the cursor is closed. Non-detached instances of that rule and instances of that rule in other detached instances, siblings, parent or child, do not have access to the persistent cursor.
  - For *Batch Rules*
  The scope is limited to the batch application. A persisted cursor opened by a particular rule is available to that rule whenever it is called within the batch application until the cursor is closed.

  The Persistent Cursor is not supported for COBOL code generation and is only available in Java.

  The host variables that are used in DECLARE ... CURSOR statements are converted before and after the SQL ASIS block containing the OPEN statement for the cursor. If DECLARE and OPEN are done in separate rule calls and host variables were modified between rule calls, then the modified values are lost. It is recommended that you only use input and global view fields as the host variables in the rules with persistent cursor.
  If cursor is not closed by the application, it is stored in the cache until the application exits (this might produce overflow) or until the same cursor is opened again.
  Restrictions and warnings concerning persistent cursor usage:

  - Cursor is persistent between different rule calls in application instance:
    - If a rule is called in different contexts (R1 calls R2 calls R3sql, but R1 also calls R3sql direct);
    - If a rule recursion is used, and the rule is already running. It is not recommended to use the persistent cursor option for such rules: cursor could be removed or overwritten by another instance of the rule.
  - Cursor is persistent for the rule in a servlet session, but does not keep persistency between different servlet sessions.
  - Cursor does not keep persistency between web service calls.
  - Cursor does not keep persistency between RMI calls.
  - Cursor does not keep persistency between different EJB sessions.
- ***Cursor declaration***: Cursors must be declared with the DECLARE CURSOR clause before the first use; otherwise, an error is issued at preparation time.
- ***Host expressions***: Host expressions can only use host variables. A host expression cannot be more complex than a single variable.
- ***FETCH statements and PRAGMA SQLCURSOR clauses***
If there are no FETCH statements in the rule and all the required table columns are not listed, then the PRAGMA SQLCURSOR clause must be used in the rule. See PRAGMA SQLCURSOR in Java.
Cursor field types are defined by types of textual first FETCH target variables or by the PRAGMA SQLCURSOR clause.
A warning is issued in the following cases:
  - If there are several FETCHes from the same cursor but types of their corresponding host variables are different, the cursor field types are defined by the first FETCH. A warning is generated on the other FETCHes with different target variables types.
  - If the cursor field types are different in the FETCH statement, the PRAGMA SQLCURSOR clause, and the PRAGMA clause

preceded ETCH, the same warnings are generated.
An error is issued in the following case:

- If there is more than one PRAGMA SQLCURSOR clause for the same cursor, even with the same list of field types, or if the PRAGMA SQLCURSOR clause is after the FETCH from this cursor.

For Java, the number of host variables in the fetch statement matches the number of columns defined for the cursor.
The number of targets in the INTO-list must be the same for all FETCH statements over one cursor. If PRAGMA SQLCURSOR is specified, then this number must coincide with the number of cursor fields specified by PRAGMA. If this condition is broken, then sqlj tool reports an error during preparation.

- ***Host variable name and cursor name***
  It is possible to use delimited identifiers (identifiers enclosed in double-quotation marks) anywhere where an ordinary identifier is allowed except for the host variable name and the cursor name. You can use a host variable even if its name is equal to a SQL reserved word without enclosing it in quotation marks. Cursor name must not coincide with any SQL reserved word.
- ***SQL ASIS block***
  The SQL ASIS block cannot contain a stored procedure declaration.
- ***Returned cursor***
  It is not possible to use a cursor returned from a stored procedure because a cursor must be declared in the same rule where it is used.
- ***Syntax for constructs***
  The syntax for the following constructs must comply with *IBM R DB2 Universal Database SQL Reference for Cross Platform Development, Version 1.1:*
  - Declare cursor statement (DECLARE...CURSOR)
  - Open statement (OPEN...)
  - Close statement (CLOSE...)
  - Fetch statement (FETCH...INTO)
- ***Syntax for SQL statements***

All other SQL statements must have syntax that is accepted by the Java SQLJ preprocessor for the installed database. For additional information see File (Database) Access Statements.

SQL CALL Statement Syntax

The following is the only syntax of the SQL CALL statement that is supported for calling a stored procedure:

```
CALL <procedure> (:[IN|OUT|INOUT]<host_variable>,...)*
```

This feature is supported only for Java.

Using Host Variables in SQL Code

In Java, SQL constructs concerning cursors are analyzed (DECLARE ... CURSOR, OPEN, CLOSE and FETCH). This enables the correct host variable conversion to be generated at the correct place. The host variables that were used in DECLARE ... CURSOR statements are converted before and after the SQL ASIS block containing the OPEN statement for the cursor.

During preparation, *:host_var = :host_var* construction is transformed to *:hostvar = (CAST :hostvar AS host_var_type)* where *host_var_type* is an SQL type corresponding to the type of the host variable. Tables of correspondence between rules and SQL data types are DBMS-specific and located in the dbms.ini file.

Automatic CAST generation can be disabled in such expressions by specifying SQLCASTOFF command-line flag; however, this can lead to run-time SQL execution errors because the DB2 SQL parser does not accept constructions such as *:host_var = :host_var* . See also Java Generation Parameters .

Example: Using Host Variables in SQL Code

```
PRAGMA SQLCURSOR (myCursor, date, time, varchar(10))

DCL
  myDate date;
ENDDCL

SQL ASIS
  Declare myCursor cursor for
  Select
  Column1,
  Column2
  From myTable
  Where Column3 = :myDate
ENDSQL

set myDate := date ()

SQL ASIS
  open myCursor
ENDSQL
```

This results in the following. The results are simplified for clarity:

```
// 0016: set myDate := date ()
fMydate.map( AbfDate.getCurrentDate() );

// 0018: SQL ASIS
/*%ConverseIn(RSQL_DATE, Mydate_sql, fMydate)%*/;
#sql [dbContext] crsMycursor =
{
SELECT
COLUMN1,
COLUMN2
FROM MYTABLE
WHERE COLUMN3 = :Mydate_sql
}
;
```

Thus, the value of myDate variable that is used in the DECLARE myCursor CURSOR statement is taken at the time of OPEN myCursor execution, not at the DECLARE ... CURSOR.

Transaction Support in Java

In Java, the Rules Language provides support for transaction management with these statements:

- START TRANSACTION
- COMMIT TRANSACTION
- ROLLBACK TRANSACTION

You can use these statements with a servlet, an Enterprise Java Bean (EJB) for a bean-managed transaction, or a Java client. For details, read these topics:

- EJB (Container to Bean) Transactions
- Client and Servlet Transaction
- Handling Rollbacks
- Clients and Database Connection Pool

EJB (Container to Bean) Transactions

The preferred mode of transaction support is the container-managed EJBs generated by AppBuilder. For normal use, do not use any of the transaction statements from the Rules Language because transactions are managed by the application server and the container.

For increased flexibility, use bean-managed transactions. Modify the transaction type in the file ejb-jar.xml from Container to Bean as follows:

```
<ejb-jar>
  <enterprise-beans>
    <session>
      ...
      <transaction-type>Bean</transaction-type>
    </session>
    ...
</ejb-jar>
```

Do not use the SQL ASIS statement to manage transactions because the application server cannot handle the transaction context. It prohibits the transactions from being propagated or processed correctly. Instead, use the Rules Language statements to start, commit, and rollback transactions.

Client and Servlet Transaction

AppBuilder supports client-managed transactions. Use this mode carefully because the transaction context in a client and server transaction exists for a longer period of time and can potentially induce deadlocks. The following restrictions apply to a full Java client:

- Set SEPARATE_RPC_THREAD=FALSE in the appbuilder.ini file.
- Do not converse any window between start and commit/rollback statements.
- Do not use detached rules.

You also need to provide information about the transaction context in the appbuilder.ini file. For example:

```
[DB]
INITIAL_CONTEXT_FACTORY=java_class_name
PROVIDER_URL=protocol://host_name:port_number
```

These are the same parameters used for an initial context in the application server.

Handling Rollbacks

Java bean-managed transactions use the transaction statements to handle rollbacks.

The container-managed transactions automatically handle rollbacks by means of the rollback flag using the SET_ROLLBACK_ONLY function. Use the GET_ROLLBACK_ONLY function to check the rollback status. It returns TRUE if the SET_ROLLBACK_ONLY function is called; otherwise, it returns FALSE.

Clients and Database Connection Pool

Connecting to a database from the application server provides a good resource management. Specify the following settings to use the database connection pool on the client side:

- Set the DB_ACCESS parameter to APPSERVER in the [DB] section of the appbuilder.ini file.
- Set the INITIAL_CONTEXT_FACTORY and the PROVIDER_URL parameters in the [DB] section of the appbuilder.ini file.
- Set the implementation name of the database object to the same name as the database connection pool name.

For additional information see File (Database) Access Statements.

Subscript Control in Java

Subscript control of occurring views is performed in Java. It relies on the INDEX_CONTROL_ABORT setting of the appbuilder.ini file. This setting controls whether or not an application aborts when a view subscript is out of range. Possible values of the INDEX_CONTROL_ABORT setting are TRUE or FALSE. If set to FALSE, then no exception is thrown, and the first occurrence is assumed if the subscript is out of range. The default value is TRUE, and the default behavior is that if a subscript is out of range error occurs, an exception is thrown, and the application terminates.

Example: Subscript Control in Java

In the following example, the `INDEX_CONTROL_ABORT` is set to FALSE, and the application continues to execute after the subscript is out of range:

```
DCL
  I INTEGER;
  V(10) VIEW CONTAINS I;
  INDX INTEGER;
ENDDCL

MAP 10 TO INDX
MAP 1 TO I(INDX) *> Correct, I(10) is set to 1 <*
MAP -1 TO INDX
MAP 1 TO I(INDX) *> Error: index less than one, first occurrence assumed <*
TRACE(I(1)) *> "1" is printed <*
MAP 11 TO INDX
MAP 2 TO I(INDX) *> Error: index greater than view size, first occurrence assumed <*
TRACE(I(1)) *> "2" is printed <*
RETURN
```

PRAGMA Statements in Java

The following PRAGMA statements have special considerations when used in Java:

- PRAGMA CLASSIMPORT in Java
- PRAGMA AUTOHANDLERS in Java
- PRAGMA ALIAS PROPERTY in Java
- PRAGMA COMMONHANDLER in Java
- PRAGMA SQLCURSOR in Java

For additional information, see Compiler Pragmatic Statements.

PRAGMA CLASSIMPORT in Java

In Java, PRAGMA CLASSIMPORT makes the fields and methods of Java classes available for the rule. An *alias* is created for this class – variable of type OBJECT TYPE 'Java class name' with a default or user-specified name. The first parameter in the list is the Javastyle class name (case-sensitive). The second parameter is the user-specified alias. If a second parameter is not used, then the default alias name (class name where the symbol '.' is replaced with the underscore symbol (_)) is created. This alias can be used to access static fields and methods of Java.

PRAGMA CLASSIMPORT Syntax

—— PRAGMA —— CLASSIMPORT ———— (—*class_alias_list* —) ————————►

where:

- *class_alias_list* is a list of pairs that consist of a Java class name and alias to import. Separate the class name and alias using a comma (spaces are ignored), and place the entire list in parentheses. The PRAGMA CLASSIMPORT clause in case-sensitive, so note the exact capitalization of the Java class name.

Whenever any class with fields or methods is used in a rule (in OBJECT TYPE '?' or OBJECT '?' declaration) a default alias is created for it. It can be changed to a more convenient name by using PRAGMA CLASSIMPORT.

Choose a unique name for an alias when using ObjectSpeak names that are the same as keywords, ObjectSpeak object types, method names, constants, or any other identifiers visible in the rule scope, to avoid ambiguity errors that might cause failures during prepare.

Example: Using PRAGMA CLASSIMPORT to Create an Alias

Alias system is created for class `java.lang.System`, and its static method exit() is invoked.

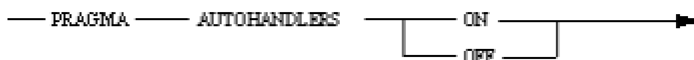
```
PRAGMA CLASSIMPORT (Customer, CUSTOMER)
PRAGMA CLASSIMPORT (Java: Entities.Employee, EMPLOYEE, Java:java.util.ArrayList, LIST_ARRAY)
PRAGMA CLASSIMPORT (Java:java.lang.System, MySystem)

dcl
  cust object type CUSTOMER;
enddcl

map new CUSTOMER to cust *> Creating a new instance <*
cust.setName("John Smith") *> Method invocation <*
MySystem.exit(0) *> Static method invocation<*
```

The Alias name assigned must be a non-existing name (in the version of Java being used). In the preceding example do not use the word `system` (assumed by Java) , but chose an unique name (as `MySystem` , etc).

PRAGMA AUTOHANDLERS in Java

In Java, PRAGMA AUTOHANDLERS determines whether or not event handlers for window objects are assigned automatically.

PRAGMA AUTOHANDLERS Syntax



The default value is ON. All handlers for window objects are assigned upon rule startup.

If PRAGMA AUTOHANDLERS OFF statement is written in a rule, these handlers are not assigned. They can be assigned later; however, using the HANDLER statement. See Event Handler Statement in Java for details.

The only exception to this rule is the INITIALIZE event of the WINDOW class (Java class `appbuilder.gui.AbfGuiWindow`). The handler for the event, if present, is enabled automatically, regardless of whether PRAGMA AUTOHANDLERS is used.

Example: Using PRAGMA AUTOHANDLERS

In this example, the window associated with a rule has two push buttons, 'OK' and 'CANCEL', and an edit field 'EDIT'.
Assume the following declarations:

```
DCL
  BUTTON_OK OBJECT 'OK';
  EDITOR OBJECT 'EDIT';
  EDITREF OBJECT TYPE EDITFIELD;
  BUTTONREF OBJECT TYPE PUSHBUTTON
ENDDCL

PROC P1 FOR CLICK OBJECT BUTTON_OK(EVT OBJECT TYPE CLICKEVENT)
...
ENDPROC

PROC P2 FOR CLICK OBJECT TYPE BUTTON(EVT OBJECT TYPE CLICKEVENT)
...
ENDPROC

PROC P3 FOR CLICK OBJECT TYPE EDITFIELD(EVT OBJECT TYPE CLICKEVENT)
...
ENDPROC

PROC INIT FOR INITIALIZE OBJECT WINDOW
...
ENDPROC
```

By default:

- `P1` is assigned for CLICK event of 'OK' pushbutton.
- `P2` is assigned for CLICK event of 'CANCEL' pushbutton.
- `P3` – for CLICK event of 'EDIT' edit field.
- `INIT` – for INITIALIZE event of a window.

If you want to use `P2` as a handler for CLICK event of 'OK' pushbutton, write:

```
HANDLER BUTTON_OK (P2)
```

If PRAGMA AUTOHANDLERS OFF is written, `P1`, `P2`, and `P3` are not assigned automatically. `INIT` still is assigned for the INITIALIZE event of the window.

PRAGMA ALIAS PROPERTY in Java

In Java, the Rules Language identifiers are *not* case-sensitive and Java identifiers are case-sensitive. Therefore, two Java class properties whose names differ only in case cannot be used directly in a rule source code. A Rule can still access these methods by declaring aliases for them. Rules Language provides the PRAGMA ALIAS PROPERTY clause for this purpose.

PRAGMA ALIAS PROPERTY Syntax

PRAGMA — ALIAS — PROPERTY — (— ' — *property_name* — ' — ,

' — *class_id* — ' | *class_name* — , — *alias* —)

where:

- *property_name* is the case-sensitive name of a property and the alias for which it is defined.
- *class_id* is a string that identifies the implementation of the class. It might be CLSID for OLE objects or the full Java class name for Java classes.The identification string is considered case-sensitive.
- *class_name* is the class name used in a rule's code. It is not case-sensitive.
- *alias* is the valid Rules identifier – alias for a method. This alias can be used in Rules code instead of the name of the method.

Example: Using PRAGMA ALIAS PROPERTY

Class `com.tinal.Data` has two fields:

- `data` of type *int* and
- `DATA` of type `java.lang.String`.

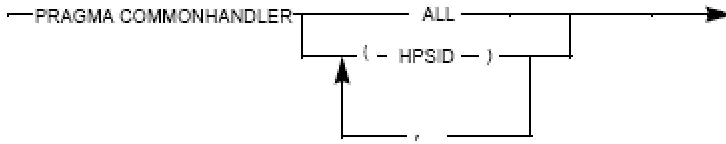
```
DCL
  D OBJECT TYPE 'com.tinal.Data';
  C CHAR(100);
  I INTEGER;
ENDDCL

set D := new 'com.tinal.Data'
MAP D.DATA TO C *> invalid: field name "DATA" conflicts with field
name "data" and can not be used directly<*

PRAGMA ALIAS PROPERTY('DATA', 'com.tinal.Data', CHARDATA)
PRAGMA ALIAS PROPERTY('data', 'com.tinal.Data', INTDATA)
MAP D.CHARDATA TO C
MAP D.INTDATA TO I
```

PRAGMA COMMONHANDLER in Java

In Java, PRAGMA COMMONHANDLER specifies the handler on any object's event using the same system ID (HPSID) within the rule scope.

For all systems IDs (HPSIDs) mentioned in the list, if an event handler is defined for the object with the same system ID (HPSID) as an object name, then this event handler is defined for all objects with the same system ID. You can either specify a list of specific system IDs to handle or specify ALL (indicates the list of all HPSIDs).

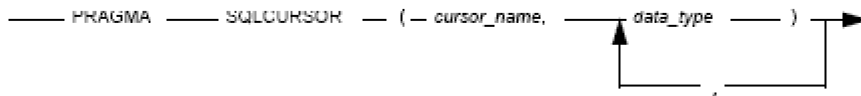
PRAGMA COMMONHANDLER Syntax



PRAGMA SQLCURSOR in Java

In Java, PRAGMA SQLCURSOR specifies cursor field types. In SQLJ, all column types must be listed when declaring an iterator (SQLJ analog for cursor). The code generation facility determines this list of types using the list of target host variables of the first FETCH from this cursor. If the cursor is not used in a FETCH statement, use PRAGMA SQLCURSOR to define this cursor's column types and avoid a preparation error.

PRAGMA SQLCURSOR Syntax



where

- *cursor_name* is any valid identifier.
- *data_type* is any primitive data type (any data type except views or objects)---see Data Types.

Example: Using PRAGMA SQLCURSOR

```
PRAGMA SQLCURSOR (myCurs, VARCHAR(255), DEC(31, 10), INTEGER)

DCL
  CurrentName VARCHAR(255);
  NewBonus INTEGER;
ENDDCL
...
SQL ASIS
  DECLARE MyCursor CURSOR FOR
  SELECT Name, Salary, Bonus
  FROM Employees
  WHERE Name = :CurrentName
ENDSQL
...
SQL ASIS
  UPDATE Employees SET Bonus = :NewBonus
  WHERE CURRENT OF MyCursor
ENDSQL
```

Static and Static Final Methods and Variables in Java

In Java, to use static and static final methods and variables of a class in a rule without creating an object of that class, use the PRAGMA CLASSIMPORT clause. See PRAGMA KEYWORD and PRAGMA CLASSIMPORT in Java for detailed syntax and examples.

Event Handler Statement in Java

In Java, the HANDLER clause enables event handlers for the specified object.

HANDLER Syntax

—— HANDLER —— *object_name* —— (— *event_handlers_list* —) —▶

where:

- *object_name* is the object variable.
- *event_handlers_list* is a list of event handlers that are delimited with commas.

Usage

By default, only event handlers for the window objects are enabled automatically. (See PRAGMA AUTOHANDLERS in Java.) For Java generation, handlers are also enabled automatically for the Rule object. Use the HANDLER statement for all other objects to enable event handlers. When a rule is terminated, all event handlers are disabled.

If any handler in the list is not defined for the object or there are several handlers for the same event and the same object, then an error is generated. Using a non-initialized object variable causes a runtime error.

Example: Enabling Event Handlers

Two event handlers of type java.awt.Button are enabled for the object of that type

```
DCL
  p object to 'java.awt.Button';
  KeyHandler1 proc for keyPressed listener
    'java.awt.event.KeyListener' type 'java.awt.Button'
        ( evt object to 'java.awt.event.KeyEvent' );
  KeyHandler2 proc for keyTyped listener
    'java.awt.event.KeyListener' type 'java.awt.Button'
        ( evt object to 'java.awt.event.KeyEvent' );
ENDDCL

HANDLER p ( KeyHandler1, KeyHandler2 )
```

Event handler is enabled for the object for which it was declared

```
DCL
  p object to 'java.awt.Button';
  KeyHandler proc for keyPressed listener
    'java.awt.event.KeyListener'
        object p ( evt object to 'java.awt.event.KeyEvent' )
ENDDCL

HANDLER p ( KeyHandler )
```

Error because KeyHandler1 and KeyHandler2 were declared for the same object

```
DCL
  p object 'java.awt.Button';
  KeyHandler1 proc for keyPressed listener
    'java.awt.event.KeyListener' type 'java.awt.Button'
        ( evt object 'java.awt.event.KeyEvent' );
  KeyHandler2 proc for keyTyped listener
    'java.awt.event.KeyListener'
        object p ( evt object 'java.awt.event.KeyEvent' );
ENDDCL

HANDLER p ( KeyHandler1,KeyHandler2 ) *> error <*
```

Error because KeyHandler was not defined for object p1

```
DCL
  p object 'java.awt.Button}}';
  p1 like p;
  KeyHandler proc for keyPressed listener
    'java.awt.event}}.KeyListener'
        object p ( evt object 'java.awt.event.KeyEvent' )
ENDDCL

HANDLER p1 ( KeyHandler) *> error <*
```

OVERLAY Statements in Java

The OVERLAY statement creates a char array from the source view or field and then interprets the char array to populate the output view or field. The following list describes how the input view fields are converted to a byte stream:

- DEC (decimal) is converted to a string representation and the string is written as character bytes. The size of the string is always length+1, meaning that the leading spaces are inserted if necessary; there are no leading zeros, and all the trailing zeros are included, while the leading sign is included only if the value is negative. The decimal separator is not included.
- PIC (picture) is converted to a string using the PICTURE format and is written as character byte stream.
- VARCHAR is interpreted as two items: the actual length and character data. The length is written first, as a smallint into two characters, and afterwards the character data. The number of characters is the maximum length of this VARCHAR. If the actual length is less then maximum, the character data is padded with blanks.
- Integer data types and data types internally represented by integer(s) (DATE, TIME, TIMESTAMP) are interpreted as bytes in little endian whose values are written to characters:
- SMALLINT as two bytes
- INTEGER as four bytes
- DATE and TIME as INTEGER (four bytes)
- TIMESTAMP is interpreted as three INTEGERS: date, time and fraction
- CHAR, DBCS and MIXED are directly written to unicode characters.

The char array created from the input view is interpreted using the structure of the output view in the following ways:

- Integer data types and data types internally represented by integer(s) (DATE, TIME, TIMESTAMP) are interpreted as bytes in little endian located in consequent characters:
- SMALLINT as two bytes
- INTEGER as four bytes
- DATE and TIME as INTEGER (four bytes)
- TIMESTAMP is interpreted as three INTEGERS: date, time and fraction
- DATE, TIME, TIMESTAMP, DEC, PIC and VARCHAR data types are validated when the data is written to the destination view, based on the data type of the target field.
- If the target field data type is DEC (decimal), the character array is read as a string of length Length(field) + 1 and converted to a decimal value using the decimal conversion routines.
- Picture (PIC) fields are read as strings and interpreted using the PICTURE format.
- VARCHAR fields are interpreted as following: maximum length(field) + 2 characters are read from the array; the first two of them are interpreted as source length (SMALLINT) and the others as field characters. The destination actual length is set to the minimum of the source length and destination maximum length. If the source length is bigger than the destination maximum length, the rest of the character is used as a source for the next field. Avoid using a VARCHAR field within the output view because of the unexpected behavior.
- The given length of a CHAR field is used to read the byte stream.

The safest way to overlay in Java is to use a target view that contains only CHAR fields, and has a total length equal to or bigger than the source view.

When the overlay source is shorter than the destination, the data in the particular field of the destination view depends on the field type and the partial data availability. If there is no data in the source to fill the field (partial data is not available), it remains unchanged, regardless of its type. If partial data is available, the result can be described by the following table:

Results depending of the available partial data

Destination view field type	If partial data are available	Example	Result

CHAR,
VARCHAR,
TEXT,
IMAGE
	Available partial data are written to the field. The rest is filled with blanks. (The behavior is different when the target field type is VARCHAR and only one character is available as partial data. In this case the field is cleared.)	`dcl`
		`  ch1 char(3);`
		`  ch2 char(4);`
		`  v1 view contains ch1;`
		`  v2 view contains ch2;`
		`enddcl`
		
		`map 'sun' to ch1`
		`overlay v1 to v2`
			`ch2 =='sun'`

SMALLINT,
INTEGER,
DEC,
PIC,
SPIC,
TIME,
DATE,
TIMESTAMP,
 	
LONGINT,
DOUBLE,
FLOAT
	The field is set to initial value.
	If APP_LEVEL constant from appbuilder.ini is more than or equal to 1, the warning message is traced.
		`dcl`
		`  ch1 char(1);`
		`  si smallint;`
		`  v1 view contains ch1;`
		`  v2 view contains si;`
		`enddcl`
		
		`map 's' to ch1`
		`overlay v1 to v2`
			Trace: Partial overlay of smallint field - field cleared
			
			`si == 0`

If the string representation of that part of the source which should be overlaid to the DATE, TIME, TIMESTAMP, DEC or PIC (SPIC) cannot be translated to the item of the destination field type, the destination field is set to the initial value.

The interpretation of the VARCHAR data item is different when overlaying directly to (or from) it:

```
OVERLAY varchar_field TO view
```

Here the `varchar_field` is interpreted as an array of its characters of length maximum `length(varchar_field)`. The varchar actual length information is not written in this case. However, for

```
OVERLAY view TO varchar_field
```

The maximum `length(varchar_field)` characters are read from the array to the fill `varchar_field` data item characters. The actual length of the `varchar_field` is set to the maximum `length(varchar_field)`. If in the array are less then the maximum length(`varchar_field`) characters, then all the characters from the array are read into `varchar_field` characters and the actual length of `varchar_field` is set to the array length.

Processing of object references in OVERLAY Statement is different from processing of fields of other types. ObjectReferences are treated as int value 0 when they are in the source view and set to null when they are the target. The result of this is that object references are not copied from source views to destination views.

Example: Processing of object references in OVERLAY statement

```
dcl
  o1, o2 object;
  v1 view contains o1, o2;
  v2 view contains o1, o2;
enddcl

map new 'appbuilder.util.AbfInt' to o1 of v1
map new 'appbuilder.util.AbfInt' to o2 of v1

overlay v1 to v2

if o1 of v2 <> o1 of v1
  trace('object references are not copied during overlay')
endif
```

In this example the output will be *object references are not copied during overlay*.

Refer to the information in the OVERLAY Statement for a comprehensive understanding of the OVERLAY statements, as the information there also applies to Java.

CASEOF in Java

In Java, it is possible to use the final fields of Java classes as selectors in CASE clauses. The final field of the Java class must be convertible to the type of the field in the CASEOF clause.

If you use the final fields of a Java class as selectors in CASEOF clause, it is not checked whether these selectors are equal to other selectors in the CASEOF statement.

USE RULE ... DETACH OBJECT Statement in Java

In Java applications, the caller rule can have limited control over the called detached rule by using the DETACH OBJECT clause. Every running rule is represented in Java by an instance of a Rule object (appbuilder.AbfModule). To access this instance, put the name of a variable of type 'OBJECT Rule' in the DETACH OBJECT clause of the USE RULE statement. When the rule is called, an instance of the called rule is assigned to a variable specified after the OBJECT keyword.

Example: Using DETACH OBJECT clause

```
DCL
  MY_CHILD_RULE OBJECT TYPE RULE;
  CHILD_WINDOW OBJECT TYPE WINDOW;
ENDDCL

USE RULE CHILD_RULE DETACH OBJECT MY_CHILD_RULE
*>CHILD_WINDOW holds reference to CHILD_RULE window<*

MAP MY_CHILD_RULE.GetWindow TO CHILD_WINDOW
```

CONVERSE REPORT Statement in Java

In Java, when you specify the printer name in a CONVERSE REPORT...PRINTER *printer_name* START statement, it overrides the corresponding appbuilder.ini setting and the report will be printed to a printer named " *printer_name* ".

For example:

```
CONVERSE REPORT MyReport PRINTER "\\server\printer" START
```

Indexed DO Statements in Java

For indexed DO Statements, native Java indexes are used in generated code whenever it is possible. Generation of native Java indexes increases performance of applications. Native Java indexes are generated when the following restrictions occur:

1. the type of the Rule loop counter is integer type (SMALLINT, INTEGER or LONGINT) or FLOAT or DOUBLE
2. the types of FROM, TO and BY expressions are integer types

The type of the native Java index depends on the type of the Rule loop counter and the types of FROM, TO and BY expressions. If the Rule loop counter is specified then the type of the native Java index will be as follows:

The type of the Rule loop counter	The type of the native index
SMALLINT	short
INTEGER	int
LONGINT	long
DEC	AbfDecimal

If the Rule loop counter is not specified then the type of the native Java index is defined by the types of FROM, TO and BY expressions:

1. if all the FROM, TO and BY expressions have SMALLINT type the type of the native Java index will be short.
2. if one of the expressions have INTEGER type and other expressions have SMALLINT or INTEGER type then the type of the native Java index will be int.
3. if one of the expressions have LONGINT type then the type of the native Java index will be long .
4. if one of the expressions have DEC type then the type of the native Java index will be AbfDecimal.

Examples: Indexed DO Statements in Java

Example1: A loop with loop counter specified

The rule

```
DCL
 counter_small SMALLINT;
ENDDCL

DO FROM 1 TO 4 BY 1 INDEX counter_small
ENDDO
```

The generated Java code for the loop

```
short i;
for(i = 1;
 i <= 4;
 i = i + 1)
{

}
fCounterSmall.map(i);
```

Example2: A loop with no loop counter

The rule

```
DO FROM 1 TO 1234567 BY 1
ENDDO
```

The generated Java code for the loop

```
for(int i = 1; i <= 1234567; i++ )
{

}
```

Native Java arrays

Java native arrays can be specified as parameters and return value when Java classes are used in the rule. For example, if we have Java classes

```
class ContainerClass
{
  public ItemClass[] getItems()
  {
    ItemClass[] items = new ItemClass[3];

    items[0] = new ItemClass("Obj1");
    items[1] = new ItemClass("Obj2");
    items[2] = new ItemClass("Obj3");

    return items;
  }

  public ContainerClass() {}
};

class ItemClass
{
  private String name;

  public ItemClass(String name)
  {
    this.name = name;
  }

  public String getName() {return name;}

};
```

then the following rule will be prepared and run successfully:

```
dcl
  containerObj object pointer to 'ContainerClass';
  arrayObj object array of object pointer to 'ItemClass';
  itemObj object pointer to 'ItemClass';
  i integer;
  err varchar(64);
enddcl

map new 'ContainerClass'() to containerObj
map containerObj.getItems() to arrayObj

do from 1 to arrayObj.Size() index i
  map arrayObj.Elem(i) to itemObj

  if i=1 and itemObj.getName() <> "Obj1"
    map err ++ "/1" to err
  endif

  if i=2 and itemObj.getName() <> "Obj2"
    map err ++ "/1" to err
  endif

  if i=3 and itemObj.getName() <> "Obj3"
    map err ++ "/1" to err
  endif
enddo

if isclear(err)
 trace("CG_RULE_LONG_NAME RESULTS: ", "SUCCESSFUL")
else
 trace("CG_RULE_LONG_NAME RESULTS: ", "ERRORS: " ++ err)
endif

return
```

Specific Considerations for CSharp

Data Types in C#

This section contains special considerations for using data types in C#. For information about data types, refer to Data Types.

- For **DATE, TIME** and **TIMESTAMP** System.DateTime type is use.
- For other type see Data Types in Java

Dynamically-Set View Functions in C#

For information about Dynamically-Set View Functions in C# see Dynamically-Set View Functions in Java

Specific Considerations for ClassicCOBOL

Specific Considerations for ClassicCOBOL

The following sections describe the specific differences in Rules Language elements for ClassicCOBOL:

- Data Types in ClassicCOBOL
- Functions in ClassicCOBOL
- OVERLAY Statements in ClassicCOBOL
- Subscript Control in ClassicCOBOL
- Procedure Declaration in ClassicCOBOL
- Double-Byte Character Set Functions in ClassicCOBOL and OpenCOBOL

- Size Limitations in ClassicCOBOL and OpenCOBOL
- Comparing Views in ClassicCOBOL and OpenCOBOL
- SETDISPLAY in ClassicCOBOL and OpenCOBOL

Restrictions on features are summarized in Restrictions on Features. To see which functions are supported, refer to Supported Functions by Release and Target Language.

Data Types in ClassicCOBOL

This section contains special considerations for using data types in ClassicCOBOL. For more information about data types, refer to the following section: Data Types.

- Decimal Field Representation in ClassicCOBOL
- Large Decimal Support for SQL in ClassicCOBOL
- DBCS and MIXED Data Types in COBOL
- Variable for the Length of the VARCHAR Data Item in ClassicCOBOL

Decimal Field Representation in ClassicCOBOL

Decimal fields up to 18 decimal digits are represented as packed decimal data. The fields with more than 18 digits are represented as PIC data. Refer to DEC in OpenCOBOL to see how decimal fields are treated in OpenCOBOL.

Large Decimal Support for SQL in ClassicCOBOL

In ClassicCOBOL, up to 31 decimal digits are supported and functions are provided when more digits are required than supported by the compiler. Code generator parameter -K defines the size of decimal fields that are generated as native COBOL packed decimal digits. All decimal fields longer than the value defined by this parameter will be generated as PIC X fields. Runtime routines perform calculations with these fields. All decimal fields used in SQL ASIS statements are passed to DB2 without any conversion.
The DDL for decimal fields larger than supported by the COBOL compiler differs from the DDL of OpenCOBOL fields. This represents a difficulty if your existing applications use large fields to convert to OpenCOBOL. Such DB2 columns, defined as CHAR are not compatible with packed decimal fields used by OpenCOBOL generation and some type of migration strategy should be considered when converting ClassicCOBOL application to OpenCOBOL.
For information about the DEC data type refer to DEC.

DBCS and MIXED Data Types in COBOL

Because of the differences in character representation on different platforms, a varied number of characters can fit into a particular MIXED field. Keep the following in mind when writing COBOL applications:
A MIXED data item's length is calculated in bytes in COBOL and can have a maximum length of 32K.
In COBOL, a single-byte character occupies one byte, a double-byte character occupies two bytes. In the beginning of a DBCS character sequence, there is a "shift-out" control character and the sequence is ended with a "shift-in" control character, each of which occupies one byte. Thus, a single DBCS character occupies up to four(4) bytes - two for character code, one for shift-out and one for shift-in.
For more information about the DBCS and MIXED data types refer to DBCS and MIXED Data Types.

Variable for the Length of the VARCHAR Data Item in ClassicCOBOL

Changing _LEN only affects the _LEN variable, the corresponding VARCHAR is *not* affected immediately. Therefore, you can use any value for the _LEN variable.
In other constructions, behavior might be different. Do not modify the VARCHAR variable through its _LEN variable in COBOL.
For more information, refer to Variable for the Length of the VARCHAR Data Item.

Example: Using _LEN variable in ClassicCOBOL

The following two examples illustrates how _LEN and the corresponding VARCHAR data item are affected by changing the _LEN.
Example 1

```
MAP -1 TO VC_LEN
MAP VC_LEN TO SomeVariable
```

SomeVariable will contain --1.

Example 2
In the following example, changing the _LEN variable does not affect the VARCHAR data item.

```
MAP "some string" TO VC
MAP 0 TO VC_LEN
IF VC = ""
    TRACE("VC is empty")
ENDIF
```

The TRACE statement will not be executed because the value of *VC* is not changed and equals " *some string* ".

Functions in ClassicCOBOL

The following functions have special considerations for ClassicCOBOL:

- Double-Byte Character Set Functions in ClassicCOBOL and OpenCOBOL
- RTRIM in ClassicCOBOL
- SETDISPLAY in ClassicCOBOL and OpenCOBOL
- STRLEN in ClassicCOBOL
- STRPOS in ClassicCOBOL
- SUBSTR in ClassicCOBOL
- UPPER and LOWER in ClassicCOBOL
- VERIFY in ClassicCOBOL

RTRIM in ClassicCOBOL

In ClassicCOBOL, if RTRIM is applied to an invalid MIXED string, the function result is undefined, but the returned string length cannot be greater than the length of the argument. For more information, see RTRIM.

STRLEN in ClassicCOBOL

In ClassicCOBOL, when STRLEN is applied to a DBCS string, the function parameter is not required to be a valid DBCS string.
If STRLEN is applied to an invalid MIXED string, the function result is undefined. It cannot be greater than the maximum length of the given string. Shift control characters that appear in COBOL only are not counted. For more information, see STRLEN.

STRPOS in ClassicCOBOL

In ClassicCOBOL, if one of the STRPOS parameters is an invalid MIXED string, the function returns zero. If the first parameter of STRPOS is a valid MIXED string and the second parameter is an invalid DBCS string, the function still returns zero. If both parameters of STRPOS are DBCS, those parameters do not have to be valid DBCS strings. For more information see STRPOS.

SUBSTR in ClassicCOBOL

In ClassicCOBOL, this function can be applied to MIXED and DBCS strings. Position and length must be specified in characters, not bytes.
In ClassicCOBOL, if SUBSTR is applied to an invalid MIXED string, the function returns an empty string. When SUBSTR is applied to a DBCS string, the function parameter is not required to be a valid DBCS string. For more information see SUBSTR.

UPPER and LOWER in ClassicCOBOL

In ClassicCOBOL, if UPPER or LOWER is applied to an invalid DBCS or MIXED string, it returns an empty string.
Characters are converted to upper case or lower case according to the specified codepage. In ClassicCOBOL, this codepage is specified by the R2C_CODEPAGE setting in the Hps.ini file and requires recompiling rules after changing it. For additional information, refer to Supported Codepages. For more information see UPPER and LOWER.

VERIFY in ClassicCOBOL

In ClassicCOBOL, if one of the VERIFY parameters is an invalid MIXED string, the function returns zero. If the first parameter of the VERIFY is a valid MIXED string and the second parameter is an invalid DBCS string, the function still returns zero. If both parameters of the VERIFY are DBCS, the parameters can contain invalid characters. For more information see VERIFY.

OVERLAY Statements in ClassicCOBOL

The OVERLAY statement in AppBuilder performs a byte-by-byte memory copy, which bypasses the MAP statement safety mechanism.
The OVERLAY statement can cause unexpected results. Although the MAP statement carefully compares view structures to make sure that data ends up only in fields like those from which it came, the OVERLAY statement blindly copies all the source data item's data, in its stored form, to the destination data item. Use caution when using OVERLAY---erroneous OVERLAY statements might not be noticed during compilation but can result in problems during execution.

Do not use MIXED or DBCS data types in OVERLAY statements.
Do not use OVERLAY statements with data types not explicitly supported.
Use MAP statements instead of OVERLAY statements whenever possible.

Refer to the information in OVERLAY Statement for a comprehensive understanding of OVERLAY statements, as the information there also applies to ClassicCOBOL.

Subscript Control in ClassicCOBOL

Subscript control of occurring views is not performed at runtime. If the subscript is out of range, a system exception occurs depending on the COBOL compiler used and the compiler's options.

Procedure Declaration in ClassicCOBOL

For ClassicCOBOL, you cannot declare a procedure return result using the LIKE clause.
For more information about the procedure, refer to Common Procedure.

Specific Considerations for OpenCOBOL

Specific Considerations for OpenCOBOL

The following sections describe the specific differences in Rules Language elements for OpenCOBOL:

- OpenCOBOL Generation
- Data Types in OpenCOBOL
- Views in OpenCOBOL
- Initialization of Occurring Views in OpenCOBOL
- Symbols in OpenCOBOL
- Functions in OpenCOBOL
- DO Statements in OpenCOBOL
- OVERLAY Statements in OpenCOBOL
- USE RULE Statement in OpenCOBOL
- PERFORM Statement (PROC) in OpenCOBOL
- PRAGMA CENTURY for OpenCOBOL
- Subscript Control in OpenCOBOL
- Native File Handling

The differences between COBOL and Rules Language lead to several restrictions to OpenCOBOL generation. These restrictions apply to built-in Rules Language functions because, in some cases, there are no corresponding COBOL functions. Also, differences in the form that data types are stored in might affect the behavior of your programs.
Restrictions on features are summarized in Restrictions on Features. To see which functions are supported, refer to Supported Functions by Release and Target Language.

OpenCOBOL Generation

OpenCOBOL is readable, maintainable and closely conforms to standard COBOL. OpenCOBOL uses standard functions and data formats where possible. OpenCOBOL does not require the use of a separate runtime for arithmetic operations with Large Decimals (meaning decimals with length more than 18) unlike those for ClassicCobol.
However, remember that this is valid only for platforms that support Large Decimals. If platform doesn't support Large Decimals, there will be syntax error concerning incorrect length of such Decimal. For specific examples of syntax errors, see Messages Reference Guide.
The following are important features of OpenCOBOL:

- ***Runtime-free code generation*** - Generated OpenCOBOL is callable without an intermediary runtime. All libraries required by the

generated COBOL are delivered in source and binary form.

- ***User-friendly code*** - Generated OpenCOBOL is readable and maintainable outside the AppBuilder environment. One externally-callable COBOL program is generated from each AppBuilder rule. Thus, there is a one-to-one relationship between AppBuilder rules and each generated COBOL program. Programs are not collapsed together.
- ***Standardized data types and functions*** - Generated OpenCOBOL uses industry-standard data types and standard COBOL functions where available.
- ***Option to prepare standard COBOL or OpenCOBOL*** - The OpenCOBOL generation facility does not replace the existing COBOL capability, but rather provides an alternative code generation option.

Data Types in OpenCOBOL

This section contains special considerations for using the following data types in OpenCOBOL. For more information about data types, refer to the following section: Data Types.

- DEC in OpenCOBOL
- LONGINT, FLOAT and DOUBLE in OpenCOBOL
- DATE, TIME and TIMESTAMP in OpenCOBOL
- Considerations for Data Items for UNIX generation
- Variable for the Length of the VARCHAR Data Item in OpenCOBOL

DEC in OpenCOBOL

OpenCOBOL supports only as many decimal digits as the compiler supports. It creates packed decimal fields in DB2 for its decimal data. Refer to Decimal Field Representation in ClassicCOBOL to see how decimal fields are treated in ClassicCOBOL.
For more information about the DEC data type, refer to DEC.

Example: Declaring DEC fields in OpenCOBOL

Given the following declarations in the rule:

```
dec_field_18_8 dec(18,8)
dec_field_25_8 dec(25,8)
```

the following declaration is used in OpenCOBOL program:

```
03 DEC-FIELD-18-8-F PIC S9(10)V9(8) USAGE COMP-3.
03 DEC-FIELD-25-8-F PIC S9(17)V9(8) USAGE COMP-3.
```

LONGINT, FLOAT and DOUBLE in OpenCOBOL

The Rules Language data types LONGINT, FLOAT and DOUBLE for (hexadecimal) floating point numbers can be translated in OpenCOBOL as follows:

LONGINT

Use LONGINT for a 64-bit signed integer:

```
PIC S9(1) USAGE COMP-4
PIC S9(1) USAGE COMP-5
```

Depending on flag CSET5 of hps.ini file.

FLOAT

Use FLOAT for single precision floating point data – value ranges from -(16^{63}-16^{57}) to 16^{63}-16^{57}:

```
USAGE COMP-1
```

DOUBLE

Use DOUBLE for a double precision floating point data – value ranges from -(16^{63}-16^{57}) to 16^{63}-16^{57}.
For more information, refer to LONGINT, FLOAT and DOUBLE in Java.

DATE, TIME and TIMESTAMP in OpenCOBOL

OpenCOBOL uses two ways to generate COBOL code for DATE, TIME and CHAR functions. One way is to generate a call to a support library routine; another is to generate COBOL code that will implement the function. Native COBOL code is generated only when second parameter, format string, for a function is a constant and this format string contains only the following tokens:

- CHAR function for DATE fields: %0m, %m, %0d, %d, %j, %0j, %c, %0c, %y, %0y, %Y
- DATE function: %0m, %0d, %0c, %0y, %Y
- CHAR function for TIME fields: %h, %0h, %t, %0t, %m, %0m, %s, %0s, %f, %0f, %x
- TIME function: %0h, %0t, %0m, %0s, %0f, %x

In all other cases support library call is generated.
OpenCOBOL uses a configurable character format. The following table compares the data types, COBOL equivalents, and the format examples for each type.

OpenCOBOL Data Type Descriptions

Data Type	COBOL Picture	Format Example
DATE	PIC X(10)	YYYY-MM-DD
TIME	PIC X(12)	HH.MM.SS.FFF
TIMESTAMP	PIC X(26)	YYYY-MM-DD.HH.MM.SS.NNNNNN

In OpenCOBOL, DATE fields are stored as PIC X(10) character fields that correspond to the DB2 configuration (e.g., yyyy-mm-dd). They are 10 bytes in length. TIME fields are stored as PIC X (12) character fields. The TIME variable has a length of 12 bytes. For more information about the DATE and TIME data types refer to Date and Time Data Types. In OpenCOBOL, TIMESTAMP fields are stored as a PIC X(26) character field in the format yyyy-mm-dd-hh.mm.ss.nnnnnn; where nnnnnn is microseconds

Considerations for Data Items for UNIX generation

When generating OpenCOBOL for HP-UX platform, the following restrictions apply:

- ***INTEGER and SMALLINT data items*** : All INTEGER and SMALLINT data items must be generated as COMP-5 by either adding FLAG=C5SET to the [CodeGen] section of the Hps.ini file or specifying -FC5SET as a code generation parameter.
- ***DEC data item*** : The length cannot be more than 18. This is a restriction of COBOL compiler and is not enforced by AppBuilder.

Variable for the Length of the VARCHAR Data Item in OpenCOBOL

Any value can be assigned to the _LEN variable.
Modify the _LEN field of VARCHAR(n) in the 0?n range in OpenCOBOL.
When the corresponding VARCHAR value is required, do not assign an invalid value (less than zero). A runtime error occurs.
If the value of the _LEN field is greater than the declared VARCHAR maximum length, the value of the corresponding variable is filled with spaces up to the actual length defined by the _LEN variable.
For more information refer to the following section: Variable for the Length of the VARCHAR Data Item.

Example: Using _LEN variable in OpenCOBOL

```
MAP -1 TO VC_LEN
MAP VC_LEN TO SomeVariable
```

`SomeVariable` will contain `-1`.

In the following example, the corresponding variable is padded with spaces to the length defined by the _LEN variable.

```
DCL
VC1 VARCHAR(5);
VC2 VARCHAR(20);
ENDDCL

MAP "12345" TO VC1
MAP 10 TO VC1_LEN
MAP VC1 TO VC2
MAP VC2 ++ "A" TO VC2
```

`VC2` will contain `'12345     A'` (five spaces before A).

Implicit Numeric Conversions in OpenCOBOL

In OpenCOBOL, values of type FLOAT are implicitly converted to values of type INTEGER by rounding decimal part (in both Cobol and Compatible arithmetic modes). This implicit conversion may occur, for example, in MAP statement or during passing of parameters.

Example: Implicit numeric conversions in OpenCOBOL

```
DCL
 f FLOAT;
 i INTEGER;
ENDDCL

PROC p(i1 INTEGER)
 TRACE(i1)
ENDPROC

MAP 1.1 to f
MAP f to i
TRACE(i)
p(f)

MAP 1.9 to f
MAP f to i
TRACE(i)
p(f)
```

In this example the output will be:

```
1
1
2
2
```

Views in OpenCOBOL

The AppBuilder code generation facility generates one copybook from each view for OpenCOBOL, which is similar to ClassicCOBOL. The AppBuilder system ID is used as the actual name of the copybook (view that is generated into COBOL). Within the copybook structure, Fields, Views, and Sets are referenced by their long names. The maximum length for a name in COBOL is 30 characters. If the View name is longer than 30 characters, AppBuilder creates a generic name. No verification on coincidence with COBOL reserved words is performed. The COBOL compiler issues an error if the identifier in the generated program is the same as a COBOL reserved word.

Initialization of Occurring Views in OpenCOBOL

Because initialization of a view causes recursive initialization of every field of that view, it can take up a large amount of resources when multiple-occurring views are used. You can prevent fields in an occurring view from being initialized by specifying a flag and an initialization setting as follows:

- Specify -FNCOCC code generation parameter flag.
- Specify OCC_VIEW_SIZE_THRESHOLD setting in the [CODEGENPARAMETERS] section of the Hps.ini file, Partitiondefault.ini for a client side code generation on the Windows platform, Unixpartitiondefault.ini for a client side code generation on the HP-UX platform, or

the codegen ini file on the host.

The integer value set for the OCC_VIEW_SIZE_THRESHOLD is used as the threshold of the occurring view size to determine whether or not the fields in the occurring views are initialized. For example, if OCC_VIEW_SIZE_THRESHOLD=299, all occurring views are initialized if the number of occurrences is less or equal to 299, but they are NOT initialized if the number of occurrences is 300 or greater.
If the flag -FNCOCC is specified and the value of OCC_VIEW_SIZE_THRESHOLD is zero or not specified, all occurring views are initialized.

You cannot reference these fields that are not initialized from outside the occurring view.

Symbols in OpenCOBOL

In OpenCOBOL, if a Set Symbol is an integer or a smallint, it can be generated with the USAGE COMP-5 clause, which is supported only in COBOL for Z/OS 3.1. To set this option, add FLAG=C5SET to the [CodeGen] section of the Hps.ini file or specify -FC5SET as a code generation parameter.
For more information, refer to Symbol.

Functions in OpenCOBOL

The following functions have special considerations when used in OpenCOBOL:

- Date and Time Functions in OpenCOBOL
- Double-Byte Character Set Functions in ClassicCOBOL and OpenCOBOL
- FRACTION in OpenCOBOL
- INCR and DECR in OpenCOBOL
- INT in OpenCOBOL
- RTRIM in OpenCOBOL
- SETDISPLAY in ClassicCOBOL and OpenCOBOL
- STRLEN in OpenCOBOL
- STRPOS in OpenCOBOL
- SUBSTR in OpenCOBOL
- TRACE in OpenCOBOL
- UPPER and LOWER in OpenCOBOL
- VERIFY in OpenCOBOL
- LONGINT, FLOAT and DOUBLE in OpenCOBOL – See Format String Specific for FLOAT and DOUBLE Data Items for more informations.

Date and Time Functions in OpenCOBOL

When the DATE and TIME functions are generated without a support library call, the results are not verified by default. The DATE function can return an invalid date value such as the 13th month. A returned TIME value can contain an invalid time, such as the 29th hour. However, you can force DATE and TIME to be validated if you specify the VERDT code generation parameter. This parameter can be set either by adding FLAG= VERDT to the [COBOL] section of the hps.ini file or by specifying -fVERDT on the PARM line under the OPENCOBOLGEN header in the CODEGEN member of the CGTABLE.
The following example illustrates the latter method:

```
&BASEQUAL.CGTABLE(CODEGEN)

[OPENCOBOLGEN]

PARM=PARAM=-VMC \-fdyncall \-yz \-fverdt
```

If `-fVERDT` is added to the code generation parameter, then if the DATE function returns an invalid date or the TIME function returns an invalid time, the returned value is converted to a special value so that the INT function returns -1 when applied to that value. When using `-fVERDT`, both DATE and TIME functions are verified.
When an integer is provided as an argument for the DATE function, it is converted to a date. When a date is provided as an argument for the INT function, it is converted to an integer. In the following example, *x* is a date and *y* is an integer:

```
DATE(INT(x))
INT(DATE(y))
```

The first expression will always equal *x* and the second expression will always equal *y* .
For general information about DATE and TIME functions, see Date and Time Function Definitions.

In cases where AppBuilder generates COBOL code for DATE/TIME functions without a call to the support library (where -fVERDT has not been set in the code generation parameter), only the format string is analyzed. The first parameter is assumed to be correct. Therefore, if the first parameter is not valid for the format string specified by the second parameter, then the result value also will be invalid. This might cause COBOL runtime to generate an exception if the value is then used in other function calls or expressions. The INT function will return an invalid result, not -1, when applied to such a value.

FRACTION in OpenCOBOL

The FRACTION function returns the fraction part with the precision available in the timestamp field. For example, if the timestamp field has the value 2003-08-21-12.53.48.976428 then the FRACTION function applied to this field returns 428000000. See Date and Time Function Definitions for more information.

INCR and DECR in OpenCOBOL

The following example illustrates how INCR and DECR functions are used in a MAP statement.

```
MAP 0 to I
MAP INCR(I) + DECR(I) + 1 to J
```

As a result, `I` is set to 0 and `J` is set to 2.

Refer to INCR and DECR in Java to see how the result is different using the same MAP statement.

INT in OpenCOBOL

In OpenCOBOL, when the INT function is applied to an invalid DATE/TIME value, it will not always return -1. See the DATE/TIME verification notes in the DATE and TIME Expressions description or Date and Time Function Definitions for more details.

RTRIM in OpenCOBOL

In OpenCOBOL, if RTRIM is applied to an invalid MIXED string, the function result is undefined, but the returned string length cannot be greater than the length of the argument. For more information, see RTRIM.

STRLEN in OpenCOBOL

In OpenCOBOL, when the STRLEN function is applied to a DBCS string, the function parameter does not have to be a valid DBCS string. If the STRLEN function is applied to an invalid MIXED string, the function result is undefined. It cannot be greater than the maximum length of the given string.
Shift control characters that appear in COBOL only are not counted. For more information, see STRLEN.

STRPOS in OpenCOBOL

In OpenCOBOL, if one of the STRPOS parameters is an invalid MIXED string, the function returns zero. If the first parameter of STRPOS is a valid MIXED string and the second parameter is an invalid DBCS string, the function returns zero. If both parameters of STRPOS are DBCS, those parameters do not have to be valid DBCS strings. For more information, see STRPOS.

SUBSTR in OpenCOBOL

In OpenCOBOL, if SUBSTR is applied to an invalid MIXED string, the function returns an empty string. When SUBSTR is applied to a DBCS

string, the function parameter is not required to be a valid DBCS string. For more information, see SUBSTR.

TRACE in OpenCOBOL

Use flag VCTRACE to instruct codegen to generate an IF condition for every VARCHAR argument of TRACE statement to verify that actual length is not zero. If it is not specified, clearing of varchar variable directly before tracing might lead to abend in run-time.

UPPER and LOWER in OpenCOBOL

In OpenCOBOL, if UPPER is applied to an invalid DBCS or MIXED string, it returns an empty string.
Characters are converted to upper case according to the specified codepage. In OpenCOBOL, this codepage is specified by the R2C_CODEPAGE setting in the Hps.ini file and requires recompiling rules after changing it. For additional information, refer to Supported Codepages. For more information see UPPER and LOWER.

VERIFY in OpenCOBOL

In OpenCOBOL, if one of the VERIFY parameters is an invalid MIXED string, the function returns zero. If the first parameter of the VERIFY is a valid MIXED string and the other is an invalid DBCS string, the function still returns zero. If both parameters of the VERIFY are DBCS, the parameters can contain invalid characters. For more information see VERIFY.

DO Statements in OpenCOBOL

A prepare error might occur if the DO statement in the Rules Language code uses values or variables that are close to the maximum size of decimal values for the target platform. This is because temporary variables are created by the Codegen to maintain the internal index. These temporary variables are allocated with enough positions to handle all possible values for the index variable.

Example: Compile Error with a Looping Structure Using Variables

The following declared variables are both within the maximum length of 18 for Unix platform; however, when they are used as in the following example, an index variable must have more than 18 digits:

```
dcl
    BIG_LEFT DEC(10,0);
    BIG_RIGHT DEC(10,10);
enddcl

do from BIG_RIGHT to BIG_LEFT
enddo
```

The resulting temporary index variable has to have a picture clause of:

```
PIC 9(10)V9(10)
```

to handle 10 decimal positions of BIG_RIGHT and 10 full digit positions of BIG_LEFT for 20 positions, which would cause a compile error if the target platform supports only 18 digits.

OVERLAY Statements in OpenCOBOL

In OpenCOBOL, the OVERLAY statement logic remains the same as in ClassicCOBOL. The only difference originates from differences in data types. Refer to specific data type descriptions in Data Types to review the differences. It is safe to overlay fields of the same data type with the same offset in source and destination views.
Refer to the information in OVERLAY Statement for a comprehensive understanding of OVERLAY statements, as the information there also applies to OpenCOBOL.

Example: OVERLAY statement in OpenCOBOL:

```
DCL
    integer_1 INTEGER;
    small_1, small_2 SMALLINT;
    date_1 DATE;

    view_1 VIEW CONTAINS small_1, small_2, date_1;
    view_2 VIEW CONTAINS integer_1, date_1;
ENDDCL

MAP DATE TO date_1 OF view_1
OVERLAY view_1 TO view_2
```

This OVERLAY statement results in date_1 OF view_2 set to the same value as date_1 OF view_1, because both DATE variables have the same offset.

If you mix data types in an overlay, the result might differ from the data types in COBOL:

```
DCL
    integer_1 INTEGER;
    small_1, small_2 SMALLINT;
    date_1 DATE;
    view_1 VIEW CONTAINS date_1, small_1, small_2;
    view_2 VIEW CONTAINS integer_1, date_1;
ENDDCL

MAP DATE TO date_1 OF view_1
OVERLAY view_1 TO view_2
```

Since DATE type representation in OpenCOBOL differs from COBOL, integer_1 OF view_2 will contain a different value than it would if you ran this example for COBOL.

USE RULE Statement in OpenCOBOL

Programs generated for OpenCOBOL and ClassicCOBOL have different calling conventions. They cannot invoke one another. The AppBuilder USE RULE statement is converted to individual program calls using a dynamic COBOL call. The HPSCOMMAREA is not used.
For more information see USE RULE Statement.

PERFORM Statement (PROC) in OpenCOBOL

The generated OpenCOBOL paragraph name is equivalent to the AppBuilder Rules Language name. The maximum length for a paragraph name in COBOL is 30 Characters, therefore, in cases where the procedure name is longer than 30 characters, AppBuilder creates a generic name. No verification is performed on coincidence with COBOL reserved words.
Recursion is not supported for procedure calls. No error message is generated when recursion is used, but execution results are unpredictable.

PRAGMA CENTURY for OpenCOBOL

In OpenCOBOL, use PRAGMA CENTURY to specify the default century used in the conversion functions DATE(char) and CHAR(date). This statement overrides the DEFAULT_CENTURY INI setting. See OpenCOBOL specific settings in the CODEGENPARAMETERS section for OpenCOBOL specific settings. PRAGMA CENTURY affects all DATE and CHAR functions that follow it until the end of the Rule code or another PRAGMA CENTURY statement. The next PRAGMA CENTURY statement overrides the previous one and behaves as described above.

PRAGMA CENTURY Syntax

PRAGMA CENTURY (string_literal)

where

- *string_literal* is any character literal containing one or two digits.

Example: Using PRAGMA CENTURY

```
DCL
    d DATE;
ENDDCL
SET d:=date("12/28/99","%0m/%0d/%0y")

//DEFAULT_CENTURY value from the INI file is used
TRACE(d)//will print 1999-12-28

PRAGMA CENTURY("18")
SET d:=date("12/28/99","%0m/%0d/%0y")
TRACE(d)//will print 1899-12-28

PRAGMA CENTURY("20")
SET d:=date("12/28/99","%0m/%0d/%0y")
TRACE(d)//will print 2099-12-28
```

Subscript Control in OpenCOBOL

Subscript control of occurring views is not performed at runtime. If the subscript is out of range, a system exception can occur depending on the COBOL compiler used and the compiler's options.

Native File Handling

This section describes the *native file handling* for OpenCOBOL on the mainframe, HP-UNIX, and AIX. This feature does not apply to Classic COBOL, Java and C# generations. The files of all types are implemented using native COBOL support. Line-sequential files are applied on the mainframe, if HFS datasets are supported.

The topics here include:

- Sequential Files Characteristics
- File Attributes
- Discriminants
- Open Function
- Close Function
- Reading operation
- Writing operation
- TRACE Function
- Changes in the Rule Hierarchy

Sequential Files Characteristics

Sequential files can be seen as a stream of bytes (an unstructured file), or as a stream of records (a structured file). The following sections discuss basic file operations. The file handling mechanism provides similar features to those currently implemented by system components. Each file comprises the following characteristics:

- Native Name
- Logical Name
- Access Mode and Physical Organization

Native Name

Each file has two names. A file has a *native name* , which links an AppBuilder file entity to the operating system's unique identifier for the file. This name depends on a target platform. For example, the native name of a file on the mainframe is the name of the DD clause, and the DD clause contains the data set name. In contrast to mainframe conventions, the file native name on the PC can be the real name of a physical file or the name of a key in the INI file, or the name of the environment variable.
In case of the DD name on the mainframe, the operating system links the DD name to the data set name before the application is started, which allows the application to operate against different physical files without the need for recompiling the application.

Logical Name

A file also has a *logical name* , which is the name given to the file within an application.

Access Mode and Physical Organization

A file also has some *access mode and physical structure* . From all the different possible file access modes, AppBuilder only supports a sequential access mode. By sequential access mode we mean that a predecessor-successor relationship among the records in the file is established by the order in which the records are placed in the file when this is created or extended.
The files with sequential access mode can have one of the several physical structures detailed below:

- Line-sequential Files
- Fixed Record Files – Files consisting of records having the same length.
- Variable Record Files – Files consisting of records having a different length.

Line-sequential Files

In case of a *line-sequential* file, each record contains a sequence of characters ending with a platform-specific record delimiter. The record delimiter is not counted in the length of the record. When the record is written, any trailing blanks are removed prior to adding the record delimiter. All the characters in the user's field from the first character up to and including the added record delimiter constitute one record and are written to the file. Upon reading the record, characters are read one at a time into the user's field until the record delimiter is encountered.
For OpenCOBOL, the line-sequential files must only contain printable characters and some control characters, for example, new-line, DBCS shift-out, DBCS shift-in, etc.
Only new-line characters are processed as record delimiters in the cases of EBCDIC encoding and ASCII encoding on UNIX, while other control characters are treated as part of the data for the records in the file. In the case of ASCII encoding on Windows, two sequential characters (carriage-return followed by new-line) are used as record delimiters. If only one of these characters is found, then it is treated as part of the data. In the case of ASCII on UNIX, one character (new-line) is used as a record delimiter.
OpenCOBOL generation control characters contains all allowable control characters for OpenCOBOL generation.

OpenCOBOL generation control characters

Control character	Hexadecimal value in EBCDIC	Hexadecimal value in ASCII
Horizontal tab	X'05	'X'09'
Vertical tab	X'0B	'X'0B'
Form feed	X'0C	'X'0C'
Carriage return	X'0D	'X'0D'
DBCS shift-out	X'0E	'X'0E'
DBCS shift-in	X'0F	'X'0F'
New-line	X'15	'X'0A'
Backspace	X'16	'X'08'
Alarm	X'2F	'X'07'

Codepage conversion is not supported for OpenCOBOL generation in the case of file operations.
The line-sequential files have no special physical characteristics to store in the model element DataSource. DataRecord elements are not used for OpenCOBOL generation.

Fixed Record Files

A file organized as *fixed* represents a sequence of records that have the same length. You can perform basic operations with these files based on a user-defined buffer. In terms of the Rules Language, such a buffer can be a VIEW or a CHAR field.
The layout of the file with a fixed structure is a sequence of records without any delimiters. Each record has the same size (as defined by the views attached to the DataRecord) and every record is treated as a sequence of bytes. The VIEW size (record length) is platform-dependent, as defined by the rules listed in Platform-specific Data Type Sizes.

Variable Record Files

A file organized as *variable* represents a sequence of records that have different lengths. You can perform basic operations with these files based on a user-defined buffer. In terms of the Rules Language, such a buffer can be a VIEW or CHAR field. The layout of the file with Variable organization is platform-dependent.
Every record of a file having variable organization on the Mainframe has control fields that precede the data. The physical record length is determined by adding 4 bytes (for the control fields) to the real record length. The record length is stored in a big-endian numeric representation. These control fields are transparent to the rule and the generated COBOL program. Thus, the OpenCOBOL generation ignores them.

File Attributes

A file is always declared by modeling it in the repository. It cannot be defined in the rule directly.
Every file entity has a *logical name* that is used in the Rules program (in terms of AppBuilder, the logical name is a long name).
Each file entity might also have a *platform specific name* (a native name in AppBuilder); otherwise, if the native name is not defined, the logical name is used as platform specific name. The platform specific name defines a DD name for the mainframe. For Windows and UNIX platforms, the platform specific name is resolved at runtime to a file name. The platform specific name might also be resolved using environment variables or INI file settings. You cannot use the platform specific file name in the rule code.
Besides the file names, the following attributes are extracted from the model, depending on the file organization type and target platform:

- The Record Length property defines maximal size of a record that can be read or written from or to the file if the Data Source

organization type is *line-sequential* or *variable*. Moreover in case of the variable organization type no attached view can have size more than this value. If the organization type is *fixed* then all attached views should have the same size that is equal to the Record Length.
- The organization type is *line-sequential* , no supplemental information is extracted from the model.
- If the type organization is *fixed* or *variable* , then all VIEWs linked to DATA_RECORDs are extracted. Also, an additional attribute from the model is extracted – inDiscriminant attributes for all relationships between fields and DataRecord (including FieldPath value).

Discriminants

There are two types of record-oriented files and two ways of reading the records from the file:

- In the first case the file contains records of the same type or the record type is known from the program/business logic before the read operation is executed.
- In the second case, some of the fields in the records need to be analyzed before the record type is known and read. In AppBuilder, such fields are marked as *discriminants* .

Files with discriminants support the following syntax to access the VIEW field that is designated as discriminant in the model:

<file_name>.<view_name1>. ... <view_nameN>.<field_name>

Some qualifiers (except *<file_name>* and *<field_name>*) can be skipped in the rule code, if they are not suspected of provoking ambiguities.

Rules Language supports discriminants using the NEXT function.

The NEXT function reads the next record into the internal buffer (that cannot be accessed directly from the program) and returns an Integer value. If the returned value is zero, then it indicates that the current reading operation is finished successfully. If the return value is greater than zero, then it indicates some exception: either end of file is encountered or reading operation was not able to finish successfully. When any read operation is performed later on a file and the buffer is not empty, then the buffer is used to populate the view and is emptied. If the buffer is empty, the data is read from the file directly into the view and the buffer stays empty.

Please note that in COBOL the buffer means the system buffer. The real emptying of the buffer does not occur, but instead the special field informs the COBOL program. Therefore, this field might have one value if the buffer is considered empty, and another if it is considered full.
If you use discriminants in the Rule code, then additional COBOL code is generated that negatively affects the application's performance.

Using discriminant field access syntax on a file with no discriminants is a syntax error.

Trying to access a discriminant when the buffer is empty (i.e. it has not been populated by the call to the NEXT function) might lead to unpredicted behavior at the execution time.

Files' usage provides two examples based on the files' type:

Files' usage

Using files with discriminant	Using files without discriminant

```
//There is a file F with three types of records:
//View1, View2, and View3
//field1 of View1 is a discriminant
//File layout:
//View1 is always followed by View3
//These pairs are intermixed with
//View2 records
//View3 that is not preceded by View1
//starts the sequence containing
//View3 records only

Open (f, READ)
Set Condition := true
//using the discriminant
Do While( Next(f) && Condition)
  Caseof (f.view1.field1)
    Case DEPOSIT
      f >> View1 //reads from buffer
// process View1
f >> View3 //reads from file
// process View3

    Case WITHDRAW
      f >> View2 //reads from buffer
//process View2

    Case Other
      Set Continue := false
Enddo
// reading the rest of the file,
// not using the discriminant

Do While (Not EOF(f))
  f >> View3 //reads from file
// process View3
Enddo
```

```
//There is a file F with three
//types of records: View1 and View2
//File layout:
//View1 is always followed by View2

Do While (Not EOF(f))
  f >> View1 //reads from buffer
// process View1
f >> View2 //reads from file
// process View2
Enddo
```

Open Function

AppBuilder supports an *Open function* with two parameters and an integer result. The first parameter is a file identifier; the second parameter defines how the file should be opened (for reading, writing, reading and writing or appending). The second parameter can be either a character literal or a symbol of the FILE_ACCESS_MODE system set; using of other constructions is not allowed. The second parameter can have one of the following values (the list below represents symbols in the new system set – File_Access_Modes; symbol names are in uppercase, symbol values are in parenthesis):

- ***READ ('r')*** – if the file should be opened to read. The Open function returns negative value if the file does not exist.
- ***WRITE ('w')*** – if the file should be opened to write, the file is created if it does not exist; it returns false if the file cannot be created or opened with WRITE access.
- ***READ_WRITE*** or ***UPDATE ('rw')*** – if the file should be opened to read and write; the file is created if it does not exist; it returns false if the file cannot be created or opened with WRITE access.
- ***APPEND ('a')*** – if the file should be opened for append; the file is created if it does not exist; it returns false if the file cannot be created or opened with WRITE access.

If the value differs from those listed above, AppBuilder reports an error.
The Open function returns zero if the current reading operation is finished successfully. If the returned value is lesser than zero, then it indicates one of the following exceptions:

- LOST_FILE (value 53). An OPEN function call with the INPUT, I-O, or EXTEND phrase was attempted on a non-optional file that was not present.
- OPENED_FILE (value 65). An OPEN function call was attempted for a file in the open mode.

It is possible to use the Open function call as a statement. In this case a value that is returned by Open function is ignored.

The examples below illustrate possible Open function uses:

Example 1:

```
if Open (MyInFile, READ) < 0
    trace ( 'It is not possible to open file', My{color:#000000}In{color}File)
    return
endif
```

Example 2:

```
Open (MyOutFile, WRITE) // return value is ignored
```

Close Function

AppBuilder supports a *Close function* with one parameter that is a file identifier. It returns an integer result. This function causes the file with the given identifier to be closed. This function returns zero if and only if the file was successfully closed. If the returned value is lesser than zero, it indicates the following exception:

- UNOPENED_FILE (value 66). A CLOSE function call was attempted for a file not in the open mode.

Example

```
if Close (MyFile) < 0
    trace ('It is not possible to close file', MyFile)
    return
endif
```

Reading operation

The binary operation >> is used to read data from a file.

Syntax

reading-operation:

file-identifier >> destination

The destination type is based on the file characteristics:

- If the file is a *line-sequential* , the destination can be CHAR, DBCS or MIXED field.
- In other cases, the destination can be either VIEW or CHAR field. If the destination is a VIEW, then it can be a view attached to a file Data Record or a view that is identical to the view attached to a file Data Record. If the destination is a CHAR field, then its size has to be equal to or greater than the size of a view or field attached to some Data Record.

VARCHAR field cannot be a destination for the read operation. This is because the VARCHAR field is generated in the COBOL program as a record containing two fields: length and storage area. If reading is performed directly to such a record, then the first two bytes of the file record (they are not record length) are copied into a length field, which might lead to an unpredictable program behavior. Supporting more sophisticated generation and setting the length field correctly might result in generated code complication, along with flexibility losses.

The reading operation returns an integer value. If the return value is zero, then it indicates that the current reading operation is successfully finished. If the return value is bigger than zero, then it indicates a possible exception: either the file's end is encountered or the reading operation was not completed successfully. These values are stored in the system error set – IO_Return_Codes. The values are:

- ***EOF*** (value -1). It indicates that the end-of-file is encountered.
- ***READ_ERROR*** (value -2). It indicates that the reading operation was unsuccessfully finished.

The reading operation returns READ_ERROR value when a problem is encountered while reading a record (for instance, the buffer size - view or field - is smaller than the file record size).

- ***LONG_RECORD*** (value -3). It indicates that the destination is shorter than the file record. Such a value can be returned when a file has variable organization only.

Semantics

The semantic of the reading operation is based on the physical file organization:

- Line-sequential Files
- Fixed Record Files
- Variable Record Files

Getting File I/O Status

AppBuilder supports a FileIOStatus function with one parameter that is a file identifier. It returns an integer result. This function does not change the file and returns current status of the file as set by the previous file operation (return value is zero if no errors happened). The status can be any of the values listed in the IO_RETURN_CODES set (see description in Native File I/O Access Modes and Status Codes section).

Example

```
if Open (MyInFile, READ in FILE_ACCESS_MODES) <> 0
    trace ('It is not possible to open file ', MyInFile)
    trace ('Error: ', SETDISPLAY (IO_RETURN_CODES, FileIOStatus (MyInFile)))
    return  {color}

endif
```

In this example file open operation is performed so we should get a trace with error message saying if file is opened.

Next Function

Function ***Next*** has one parameter, which is a file; it returns an integer value, similar to a reading operation described below. If the next record does not exist or is shorter than a DataRecord size (in case of some corrupt files), then EOF value is returned.

When this function is executed it performs read operation as described below but the record read from a file is stored in the internal buffer generated in the COBOL program. Rule code has no access to this buffer but can analyze discriminant fields after NEXT was executed.

The read operation for the files with discriminant can be used with or without the call to the NEXT function. The semantics of the reading operation are the following:

If the *buffer is empty*, then the reading operation reads the record from the file to a destination; the buffer remains empty.

If the *buffer is not empty*, then the reading operation moves the buffer content to a destination; this operation empties the buffer.

See Read Operation below for more details and examples.

Read operation

The binary operation >> is used to read data from a file.

Syntax

read_operation:

  file_identifier >> destination

The destination type is based on the file characteristics:

- If the file is a *line-sequential file*, the destination can be CHAR, DBCS or varchar field.
- In other cases, the destination can be VIEW, CHAR, DBCS or varchar field.
- If the destination is a VIEW, then it can be a view attached to a file Data Record or a view that is identical to the view attached to a file Data Record.
- If the destination is a CHAR field, then its size has to be equal to or greater than the size of a view or field attached to some Data Record.
- If destination is a VARCHAR field then its size has to be equal to or greater than the size of a view or a field attached to some Data Record. A length of the VARCHAR field is set to the size of a read record.

The read operation returns an integer value.

- If the return value is greater than zero or is equal to zero indicates that the current read operation is successfully and this value means a length of the read record.
- If the return value is less than zero, then it indicates a possible exception: either the file's end is reached or the read operation is not completed successfully. This values are defined in the system set IO_Return_Code (see description in Native File I/O Access Mode and Status Codes section):
    - EOF
    - UNOPENED_FILE
    - LONG_RECORD

- INCOMPLETE_RECORD
- NON_READ_MODE

Semantics

The semantic of the read operation is based on the physical file organization:

- Line-sequential Files
- Fixed Record Files
- Variable Record Files

Line-sequential Files

If a file has a *line-sequential* organization, then the characters in the file record are read one at a time into the destination until one of the following conditions occurs:

- The *record delimiter* (new-line character) is encountered. The delimiter is discarded. The remainder of the destination is filled with spaces (*the destination is longer than the file record*). It returns the number of bytes actually read from the file, without the record delimiter(s).
- The *end of the destination* has been reached. The entire target record area is filled with characters. If the next unread characters are the record delimiters, they are discarded. The next >> operation reads the first character of the next record (*the destination has the same length as the file record*). It returns the number of bytes actually read from a file, without the record delimiter(s).
- Otherwise *the destination is shorter than the file record* . The current reading operation is completed and the next unread character from the current file record is the first character to be read by the next >> operation. In this case, the reading operation returns the LONG_RECORD value.
- The *end-of-file* is encountered. The remainder of the destination is filled with spaces (*the destination is longer than the file record*). It returns the number of bytes actually read from a file, without record delimiter(s).

The file codepage attribute is ignored and there is no codepage conversion while reading the line-sequential file.

Fixed Record Files

If a file has a *fixed* organization, then DATA_SOURCE has more than one DATA_RECORD entity describing one of the record formats that can be stored in the file. Every DATA_RECORD is linked to a single VIEW object. The right operand of the reading operation could be any view in the rule scope or any local view or any CHAR field.
The semantics of a reading operation are discriminant-dependent:

- If a file has *no discriminants* , then the reading operation inputs the next record from a file to a destination, as shown in Example 1: File Without Discriminants.
- If a file has *discriminants* , then the rule code might first access a discriminant(s) to make a choice of concrete view that becomes the reading operation's destination, as shown in Example 2: File With Discriminants.
- The program *reads the next record* from a file without specifying the destination. This operation is implemented as reading the next record into some internal buffer.
- The program *analyzes discriminants* . This is implemented as reading from the buffer, transparent to the rule.
- The program *reads the record* into a discriminant value-based chosen view. This is implemented as Overlay of the buffer into the specific view.

In this case, the reading operation is not sufficient. It is necessary to have one additional function that reads the next physical record into an internal buffer:

- function Next has one parameter, which is a file; it returns an integer value, similar to a reading operation. If the next record does not exist or is shorter than a DataRecord size (in case of some corrupt files), then EOF is returned.
- a buffer, for storing the record, generates and it is initialized when the file is open. In addition, it allows accessing the fields declared as discriminants.

The reading operation can also be used without the preceding call to the NEXT function. In this case, the semantics of the reading operation are the following:

- If the *buffer is empty* , then the reading operation reads the record from the file to a destination; the buffer remains empty.
- If the *buffer is not empty* , then the reading operation moves the buffer content to a destination; this operation empties the buffer.

All buffer operations are transparent to the program and the program does not have direct access to the buffer. Only the Next and >> reading operations might implicitly access or change the internal buffer; discriminant access might also access the internal buffer.

Example 1: File Without Discriminants

There is a fixed record file MyInFile with two types of records: View1 and View2 with the following file layout: View1 is always followed by View2. The program should read this file and process its records.

```
dcl
    l_view1 like view1;
    l_view2 like view2;
    IO_RC integer;
enddcl

if Open (MyInFile, Read in File_Access_Mode) < 0
    trace (‘File MyInFile cannot be open’)
    trace (MyInFile)
    return
endif

do set IO_RC := MyInFile >> l_view1
    if IO_RC >= 0
        ProcessView1 (l_view1)
        set IO_RC := MyInFile >> l_view2{color}
    endif
while IO_RC >= 0
    ProcessView2 (l_view2)
enddo

if IO_RC < 0 and IO_RC <> EOF in IO_Return_Codes
    trace ('File Read Error:', SETDISPLAY(IO_Return_Codes, IO_RC))
endif
```

Example 2: File With Discriminants

There is a fixed record file MyInFile with two types of records: View1 and View2. One of the records has a discriminant: the field MyDiscrView1 is discriminant of View1; View2 has no discriminant. The program should read this file and process with its records.

```
dcl
    l_view1 like view1;
    l_view2 like view2;
    read_ret_code integer;
enddcl

if Open (MyInFile, Read in File_Access_Mode) < 0
    trace (‘File MyInFile cannot be open’)
    trace (MyInFile)
    return
endif

do set read_ret_code := Next (MyInFile)
while read_ret_code = 0
    if MyInFile.View1.MyDscrView1 = ‘YES’
        set read_ret_code := MyInFile >> l_view1
        // reading from a buffer does not generate any errors
// so error code is not analyzed
ProcessView1 (l_view1)
    else
        MyInFile >> l_view2
        ProcessView2 (l_view2)
    endif
enddo
```

EndOfFile condition

To check whether the file reading operation have reached end of file, Rules Language supports EOF function that returns TRUE if and only if the end of the file had been reached by the previous reading operation.

```
// This is a file f with records of View1 type
do while FileIOStatus (f) = 0
    f >> View1   // reads from buffer
// process View1 for odd records here
if FileIOStatus (f) = 0
        f >> View1  //reads from buffer
// process View1 for even records here
else
        if EOF (f)
            trace ('Warning: EOF - no more records in the file.')
        endif
   endif
enddo
```

Variable Record Files

If a file has a *variable* organization, then the file's next record is read one at a time into the destination until the end-of-file is encountered. If the record has a length greater than the destination, then the remainder is discarded and the reading operation returns LONG_RECORD value from IO_Return_Codes.
The semantic of the reading operation is identical to the OVERLAY statement, where the source is the CHAR field with the same size as the file record and the destination is a view or a field (for instance, the bytes from a file record are copied one by one into the memory area allocated by the destination).
The file that has variable organization can also have discriminants. The discriminants' support is the same as for fixed organization files.

Write operation

The binary operation << is used to write data to a file.

Syntax

writing-operation:

file-identifier << source

The source contains the data that should be written to a file. The type of the source depends on the file characteristics:

- If a file has a *line-sequential* organization, then the source can be a CHAR, DBCS, or MIXED field.
- In case of *fixed or variable* organization, the source can be either a VIEW or a CHAR/DBCS/MIXED field.

The write operation returns an integer value. If the returned value is zero, then it indicates that the current reading operation is finished successfully. If the returned value is lesser than zero, then it indicates an exception. This values are defined in the system set IO_Return_Code (see description in Native File I/O Access Mode and Status Codes section).

Semantics

If a file has a *line-sequential* organization, then the new record is placed immediately after the last record written to the file by the previous write operation. If the source is a CHAR/DBCS field, then the spaces (single byte and DBCS) at the end of the source are discarded, and the record delimiter is added at the end of the source. The characters in the source from the first character up to and including the added record delimiter are written to the file as one record.
If a file has a *fixed or variable* organization, then the new record is placed immediately after the last record written to the file by the previous write operation.
For fixed or variable organization, if the source is a CHAR field, then the entire field content is written to the file. In all other cases the source is considered the sequence of bytes and all the bytes in the source are written to the file.
If the file was open for the *Read operation* , then it causes an exception. If the file is open for the *Write operation* , then the first write operation writes a record to a file at offset 0. If the file is open for *Append operation* , then the first write operation puts a record after the last record that already exists in the file.

TRACE Function

You can use the TRACE function with a file identifier as a parameter. The output consists of information about a file (for example, logical and native name, organizationType, etc.).

Support of Global Data Source

Data Sources can be shared between several rules. To support this feature it needs to mark as Global on the relationship between a rule and the Data Source then such Data Source is shared between all the rules in the same transaction that also have the Global attribute for this Data Source.

Native File I/O Access Mode and Status Codes

This paragraph contains a description of two new system sets that can be used to support a native file handling.

Table 13-10. FILE_ACCESS_MODE set

This set provides values to use them as second parameter of Open function call.

Set Symbol Name	Set Symbol Value
READ	r
WRITE	w
UPDATE	rw
APPEND	a

Table 13-11 IO_RETURN_CODES set

This set is used to check error codes returned by Input/Output operations.

Set Symbol Name	Set Symbol Value
SUCCESS	0
EOF	-1
DATA_SOURCE_ERROR	-2
READ_ERROR	-3
OPEN_MODE_ERROR	-4
OPEN_ERROR	-5
LONG_RECORD	-6
NON_EXISTING_FILE	-7
OPEN_FILE	-8
UNOPEN_FILE	-9
NON_READ_MODE	-10
INCOMPLETE_RECORD	-11
WRITE_ERROR	-12
NON_WRITE_MODE	-13
IO_ERROR	-128

Changes in the Rule Hierarchy

If a rule contains in its hierarchy a top-level view, then such a view cannot be used as a child of any other view in the rule scope. For example, if a rule has a window attached and the window has a view attached, then such a view is a top-level view. As a result, such a view cannot be used as a sub-view of the rule input view, or sub-rule output view or any other view that is visible to the rule.
Such a restriction is not enforced by the hierarchy, but by Codegen, because if a view is used as a sub-view of another view, then it no longer exists in the rule hierarchy as a separate entity. If such a view were created as a separate entity, then it would be impossible to access its' fields, and codegen would report such a reference as ambiguous.
This restriction creates certain design difficulties or inefficiencies for AppBuilder applications, especially for files. For example, if a view OrderRecord is linked with a DataRecord, and the user wants to return such a record in the output view, then the only solution is to define view Order with a structure identical to view OrderRecord and attach the view Order to the output view. In the rule code, the record can be read directly to view B:

MyFile >> Order

This requires a duplicated view structure in the repository and, therefore Codegen generates two COBOL records (and two Java classes), with an initialization code for both of them, although OrderRecord is not even used in the rule.
An alternative is to attach view OrderRecord to the DataRecord and as a subview of the output view. Codegen allows this, but with one condition (which is already used for the views that are declared in the rule DCL section): if any top-level view from the rule scope is used as a sub-view of any other top-level view, then such a view is no longer generated as a separate entity. In the example above, the view Order is no longer necessary and the view OrderRecord can be directly attached to the output view and, subsequently, used in the rule code:

MyFile >> OrderRecord

The existing restriction cannot be removed for all views in the rule scope:

- the rule input or output views cannot be used as sub-views of any other view;
- the sub-rules' input/output views can be used, but it might require significant changes in the generation;
- the global view can not be used.

Specific Considerations for ClassicCOBOL and OpenCOBOL

Specific Considerations for ClassicCOBOL and OpenCOBOL

The following sections describe the specific differences in the Rules Language elements that apply to both ClassicCOBOL and OpenCOBOL:

- Size Limitations in ClassicCOBOL and OpenCOBOL
- Comparing Views in ClassicCOBOL and OpenCOBOL
- Double-Byte Character Set Functions in ClassicCOBOL and OpenCOBOL
- SETDISPLAY in ClassicCOBOL and OpenCOBOL
- User Components in ClassicCOBOL and OpenCOBOL

Size Limitations in ClassicCOBOL and OpenCOBOL

Data items larger than 16777215 bytes are not allowed. This same limitation also applies to the number of occurrences.
For more information, refer to View.

Comparing Views in ClassicCOBOL and OpenCOBOL

View comparison is implemented as a native COBOL record comparison. Because view comparison does not take into account the data type of the fields in the view, it is possible for the comparison of two views to give a different result than the comparison of the fields in the view.
For more information, refer to Comparing Views.

Double-Byte Character Set Functions in ClassicCOBOL and OpenCOBOL

The following information pertains to the implementation of double-byte character set functions in ClassicCOBOL and OpenCOBOL:

CHAR function

- With a CHAR argument, the CHAR function returns its argument.
- With a MIXED argument, the CHAR function returns the argument that concatenates the deleted empty DBCS portions and successive DBCS portions. This process is called normalization. The string returned has the same length, but if it has been shortened by normalization, it is padded with single-byte spaces.
- With a DBCS argument, the CHAR function returns the argument with shift control characters added before (a "shift-out" character) and after (a "shift-in" character) the DBCS string.

MIXED function

- With a CHAR and a MIXED argument, the MIXED function returns a normalized argument.
- With a DBCS argument, the MIXED function returns the argument with shift control characters added before ("shift-out" character) and after ("shift-in" character) given by the DBCS string.

DBCS function

- With a CHAR and a MIXED argument, the DBCS function assumes that the first and last characters of a given CHAR string are "shift-out" and "shift-in" control characters respectively, and returns the argument with the first and last characters deleted.
- With a DBCS argument, the DBCS function returns its argument.

Some of these functions perform Validation and Implementation of Double-Byte Character Set.

SETDISPLAY in ClassicCOBOL and OpenCOBOL

In ClassicCOBOL and OpenCOBOL, the RTRIM function is required for DBCS display if the display length is less than 39 characters (78 bytes

plus 2 shift characters). If the display length is less than 39 characters, the returned value is padded with single-byte characters. If the returned value that has been padded with single-byte characters is used as an argument in the SETDISPLAY function, the SETDISPLAY function returns all spaces because the value with different types of trailing spaces or shift characters sequences are considered non-equal. See DBCS and MIXED Data Types and SETDISPLAY for details.

User Components in ClassicCOBOL and OpenCOBOL

ClassicCOBOL calls user components using the HPSCOMMAREA. In OpenCOBOL, the generated calling convention is optional to avoid changing every user component. The global flag is used during COBOL generation to specify the generation of the HPSCOMMAREA or a dynamic COBOL call. AppBuilder provides flag settings according to the component's target environment (CICS or BATCH).
AppBuilder generates the call differently for each target depending on how the user component accepts its parameters. The parameters are global, so if you generate a component call, all the receiving components in the hierarchy must be coded in the same manner.
Each target environment flag has three settings:

- Y Setting
- N Setting
- B Setting

Y Setting

The Y setting passes the parameters DFHEIBLK and HPSCOMMAREA, as it is done in standard COBOL. Using this setting ensures that all previously written components will work as before. AppBuilder populates the input/output view address. To retrieve the address of working storage, AppBuilder generates a contained program that receives the working storage view as a parameter and then uses ADDRESS OF to pass the pointer back to the calling rule. This pointer then populates the INPUT/OUTPUT view pointer in the commarea.
The contained program is embedded in the main program and delimited by the IDENTIFICATION DIVISION and END PROGRAM statements. A separate contained program must be built for each view address needed.

Calling Rule:

CALL 'ABIOADDR' INPUT-VIEW, INPUT-VIEW-PTR

```
MOVE VIEW-ADDRESS OF INPUT-VIEW TO IV-ADDRESS OF RULE-COMP-COMMAREA.
MOVE VIEW-ADDRESS OF OUPUT-VIEW TO OV-ADDRESS OF RULE-COMP-COMMAREA.
CALL 'ABCDEF' USING DFHEIBLK, RULE-COMP-COMMAREA.
IDENTIFICATION DIVISION.
PROGRAM-ID. ABIOADDR.
DATA DIVISION.
LINKAGE SECTION.
01 VIEW-ADDRESS-PTR POINTER.
01 INPUT-VIEW.
03 IN-DATA PIC X(80).
PROCEDURE DIVISION USING VIEW-ADDRESS-PTR, INPUT-VIEW.
SET VIEW-ADDRESS-PTR TO ADDRESS OF INPUT-VIEW.
END PROGRAM JMDADDR.
```

Called Component:

PROCEDURE DIVISION USING DFHEIBLK, RULE-COMP-COMMAREA.

N Setting

The N setting passes the INPUT and OUTPUT view as parameters. Use the same parameters for the calling components PROCEDURE DIVISION statement. If you are calling CICS components that initiate CICS calls, ensure that the CICS translator uses the NOLINKAGE option. The component must establish addressability to the EIB.

Calling Rule:

CALL 'ABCDEF' USING INPUT-VIEW, OUTPUT-VIEW

Called Component:

PROCEDURE DIVISION USING INPUT-VIEW, OUTPUT-VIEW

B Setting

The B setting passes a surrogate DFHEIBLK and DFHCOMMAREA along with an input/output view. AppBuilder needs this option for components that run through the CICS translator, which by default puts the DFHEIBLK and DFHCOMMAREA on the PROCEDURE statement and LINKAGE SECTION. The Commarea creates an artificial 1 byte parameter.

Calling Rule:

CALL 'ABCDEF' USING DFHEIBLK, RULE-COMP-COMMAREA, INPUT-VIEW,OUTPUT-VIEW.

Called Component:

PROCEDURE DIVISION USING DFHEIBLK,RULE-COMP-COMMAREA,INPUT-VIEW,OUTPUT-VIEW

Restrictions on Features

This section describes restrictions on features that apply to all AppBuilder releases regardless of target languages, and restrictions that apply to specific target languages.

- Restrictions on Features by Release
- Restrictions on Features by Target Language

Restrictions on Features by Release

The following table lists restrictions on features for applicable releases that apply to all target languages unless specifically stated.

Restrictions specific to releases

Release	Restrictions
AppBuilder 3.2	Not DBCS-certified Macro support is available on the PC (client side), and on the mainframe; you can still prepare C to the host (UNIX) by running macro expansion on the client.
AppBuilder 3.2 Host	Not DBCS-certified
AppBuilder 2.0.3.5 Host	PRAGMA SQLCURSOR is not supported.

Restrictions on Features by Target Language

The following table summarizes restrictions on features by target language that apply to all AppBuilder releases. Blank cells in the table below indicate no restrictions for that language.

Restrictions on Features by Target Language

Feature	C	Java	.NET	ClassicCOBOL	OpenCOBOL
3270 converse mainframe system components			Not supported		Not supported
COMMIT TRANSACTION statement			Not supported		Not supported
CONVERSE statement		Not supported	Not supported		Not supported
CONVERSE WINDOW statement		Not supported	Not supported		Not supported
CONVERSE REPORT statement	Not supported		Not supported		Not supported
Code generation	Not DBCS certified		Not DBCS certified		
DBCS and MIXED parameters with string functions	Not supported		Not supported		

DBCS and MIXED data			Not supported		No data validation is performed. Not supported for OpenCOBOL client side generation, except AIX.
DBCS object names			Not supported		Not supported [1]
DBCS codepage			Not supported		OpenCOBOL support library is not customized for any DBCS codepage. Customizing ABNLS C module will provide the required functionary.
Decimal arithmetic CALCULATOR mode	Used as default.	Used as default.	Used as default	Not supported	Not supported
Decimal Arithmetic COMPATIBILITY mode		Not supported	Not supported		Not supported
Decimal arithmetic COBOL	Not supported	Not supported	Not supported	Used as default.	Used as default.
Decimal fields				Cannot be used with the TRACE function. If a decimal field is used, a code generation error message is received during preparation.	
Dynamic occurring views	Not supported			Not supported	Not supported
Escape sequences \a, \v, ?		Not supported	Not supported		
FRACTION					Returns microseconds only, See FRACTION in OpenCOBOL for details.
Global views		Only within single JVM instance	Only within single .NET machine or application instance		
LIKE clause					
OBJECT data type, ObjectSpeak and all related statements (NEW, method calls, etc.)	Deprecated			Not supported	Not supported
object aliases (see *Alias* for information about the Aliases object data item)	Deprecated			Not applicable	Not applicable
OVERLAY statement (Data types representation is platform specific, which may result in different results for an overlay statement.)					
Procedure calls recursion				Not supported for procedure calls. No error message generated if used, but execution results will be unpredictable	Not supported for procedure calls. No error message generated if used, but execution results will be unpredictable

ROLLBACK TRANSACTION statement			Not supported		Not supported
Set Define values containing DBCS characters			Not supported		Not supported [2]
Set symbol	Decimal part is truncated if it is longer than declared in the set.	If set symbol is inconsistent with the set definition, the code generator will generate an error.	If set symbol is inconsistent with the set definition, the code generator will generate an error. If static set symbol is changed, then all the rules that use such symbol must be re-prepared.		Decimal part is truncated if it is longer than declared in the set.
SETDISPLAY and SETENCODING					Accept only LOOKUP and ERROR sets as parameters
START TRANSACTION statement			Not supported		Not supported
subscript control		No subscript control at compile time because of dynamic views support	No subscript control at compile time because of dynamic views support		
TIMESTAMP					Precision is up to the microseconds only.
TIME, DATE, and TIMESTAMP					Representation is different from all other platforms, which might affect some of the statements. OVERLAY is one example of an affected function.

[1] COBOL only allows the use of the Roman alphabet for program identifiers. Therefore, DBCS and MIXED identifiers used in Rules code are not supported. The preparation will not issue any error messages. However, the COBOL compiler issues error messaged if the generated COBOL code is not correct.

[2] If Set Define values contain DBCS characters, this results in COBOL names containing DBCS and non-DBCS characters, which is not supported by the IBM COBOL compiler and results in a COBOL compiler error. If Define values contain only DBCS characters, the OpenCOBOL generation option GENNOSUFF can be used to prevent the generation of mixed COBOL identifiers. Avoid the use of DBCS characters for define values. When GENNOSUFF is used, use it for the entire application. When defining object names, avoid the generation of COBOL reserved words, such as RETURN-CODE.

Supported Functions by Release and Target Language

The following table summarizes supported functions by release and target language. The target languages indicate where the code is generated. Functions marked with an X in the table are supported in that release for that target language. A blank cell indicates the function is not supported.

Supported Functions by Release and Target language

Function	**Target Language (X means supported)**							
	AB2035 Host		**AB32**					

	Classic COBOL	Open COBOL	C	Java	.NET	OpenCOBOL		
						Host	HP-UX	AIX
ADDR(view)	X	X				X	X	X
APPEND				X	X			
DELETE				X	X			
INSERT				X	X			
REPLACE				X	X			
RESIZE				X	X			
OCCURS	X	X	X	X	X	X	X	X
SIZEOF	X	X	X	X	X	X	X	X
CLEARNULL				X	X			
ISNULL				X	X			
HIGH_VALUES	X	X	X	X	X	X	X	X
LOW_VALUES	X	X	X	X	X	X	X	X
TRACE	X	X	X	X	X	X	X	X
SETDISPLAY	X	X	X	X	X	X	X	X
SETENCODING	X	X	X	X	X	X	X	X
CEIL	X	X	X	X	X	X	X	X
FLOOR	X	X	X	X	X	X	X	X
ROUND	X	X	X	X	X	X	X	X
TRUNC	X	X	X	X	X	X	X	X
INCR[1]	X	X	X	X		X	X	X
DECR[2]	X	X	X	X		X	X	X
CHAR(integer)	X	X	X	X	X	X	X	X
CHAR(char)	X	X	X	X	X	X	X	X
CHAR(dbcs)	X	X	X	X	X	X		X
CHAR(mixed)	X	X	X	X	X	X		X
CHAR(dec)	X	X	X	X	X	X	X	X
CHAR(integer, char)	X	X	X	X	X	X	X	X
CHAR(dec, char)	X	X	X	X	X	X	X	X
CHAR(dec, char, char)			X					
CHAR(time)	X	X	X	X	X	X	X	X
CHAR(date)	X	X	X	X	X	X	X	X
CHAR(date, char)	X	X	X	X	X	X	X	X
CHAR(date,mixed)	X			X	X			X
CHAR(time, char)	X	X	X	X	X	X	X	X
CHAR(time, mixed)	X			X	X			X
CHAR(timestamp)		X	X	X	X	X	X	X

CHAR(timestamp, char)		X		X	X	X	X	X
DBCS(dbcs)	X	X	X	X				X
DBCS(char)	X	X		X				X
DBCS(mixed)	X	X		X				X
MIXED(mixed)	X	X	X	X				X
MIXED(char)	X	X	X	X				X
MIXED(dbcs)	X	X	X	X				X
DATE	X	X	X	X		X	X	X
DATE(dbcs)	X	X		X				X
DATE(dbcs,char)	X	X		X				X
DATE(dbcs,mixed)	X	X		X		X		X
DATE(timestamp)	X	X	X	X	X	X	X	X
DATE(char)	X	X	X	X	X	X	X	X
DATE(integer)	X	X	X	X	X	X	X	X
DATE(char,char)	X	X	X	X	X	X	X	X
DATE(mixed)	X	X		X				X
DATE(mixed,char)	X	X		X				X
DATE(mixed,mixed)	X	X		X				X
TIME	X	X	X	X	X	X	X	X
TIME(integer)	X	X	X	X	X	X	X	X
TIME(dbcs)	X	X		X				X
TIME(dbcs,char)	X	X		X				X
TIME(dbcs,mixed)	X	X		X		X		X
TIME(mixed)	X	X		X				X
TIME(mixed,char)	X	X		X				X
TIME(mixed,mixed)	X	X		X				X
TIME(timestamp)	X	X	X	X		X	X	X
TIME(char)	X	X	X	X	X	X	X	X
TIME(char,char)	X	X	X	X	X	X	X	X
TIMESTAMP	X	X	X	X	X	X	X	X
TIMESTAMP(char)		X		X	X	X	X	X
TIMESTAMP(date,time,integer)	X	X	X	X	X	X	X	X
HOURS(time)	X	X	X	X	X	X	X	X
MONTH(date)	X	X	X	X	X	X	X	X
YEAR(date)	X	X	X	X	X	X	X	X
DAY(date)	X	X	X	X	X	X	X	X
DAY_OF_YEAR(date)	X	X	X	X	X	X	X	X
DAY_OF_WEEK(date)	X	X	X	X	X	X	X	X
MILSECS(time)	X	X	X	X	X	X	X	X

MINUTES(time)	X	X	X	X	X	X	X	X
MINUTES_OF_DAY	X	X	X	X	X	X	X	X
NEW_TO_OLD_DATE(date)	X	X	X	X	X	X	X	X
NEW_TO_OLD_TIME(time)	X	X	X	X	X	X	X	X
OLD_TO_NEW_DATE(integer)	X	X	X	X	X	X	X	X
OLD_TO_NEW_TIME(integer)	X	X	X	X	X	X	X	X
SECONDS(time)	X	X	X	X	X	X	X	X
SECONDS_OF_DAY(time)	X	X	X	X	X	X	X	X
INT(char)	X	X	X	X	X	X	X	X
INT(time)	X	X	X	X	X	X	X	X
INT(date)	X	X	X	X	X	X	X	X
INT(char,char)	X	X	X	X	X	X	X	X
INT(char,char,char)			X		X			
DECIMAL(char)	X	X	X	X	X	X	X	X
DECIMAL(char,char)	X	X	X	X	X	X	X	X
DECIMAL(char,char,char)			X		X			
FRACTION(timestamp)	X	X	X	X	X	X	X	X
GET_ROLLBACK_ONLY				X				
SET_ROLLBACK_ONLY				X				
HPSCOLOR(integer)			X					
RGB(integer,integer,integer)			X	X				
HPSERROR			X					
HpsResetError			X					
HpsErrorMessage(integer)			X					
LOC(view):CHAR	X	X	X			X	X	X
LOC(view):OBJECT	X	X	X	X	X	X	X	X
LOC(integer):OBJECT	X	X	X	X	X	X	X	X
LOC(smallint):OBJECT	X	X	X	X	X	X	X	X
LOC(char):OBJECT	X	X	X	X	X	X	X	X
LOC(dbcs):OBJECT	X	X	X	X	X	X	X	X
LOC(mixed):OBJECT	X	X	X	X	X	X	X	X
LOC(dec):OBJECT	X	X	X	X	X	X	X	X
LOC(date):OBJECT	X	X	X	X	X	X	X	X
LOC(time):OBJECT	X	X	X	X	X	X	X	X
LOC(timestamp):OBJECT	X	X	X	X	X	X	X	X
LOC(boolean):OBJECT	X	X	X	X	X	X	X	X
LOC(object):OBJECT	X	X	X	X	X	X	X	X
VERIFY(char,char):smallint	X	X	X	X	X	X	X	X
VERIFY(dbcs,char)	X	X		X				X

VERIFY(mixed,char)	X	X		X				X
VERIFY(mixed,mixed)	X	X		X				X
VERIFY(mixed,dbcs)	X	X		X				X
LOWER(char):char	X	X	X	X	X	X	X	X
LOWER(dbcs):dbcs	X	X		X				X
LOWER(mixed):mixed	X	X		X				X
UPPER(char):char	X	X	X	X	X	X	X	X
UPPER(dbcs):dbcs	X	X		X				X
UPPER(mixed):mixed	X	X		X				X
STRLEN(char):smallint	X	X	X	X	X	X	X	X
STRLEN(dbcs):smallint	X	X		X				X
STRLEN(mixed):smallint	X	X		X				X
STRPOS(char,char):smallint	X	X	X	X	X	X	X	X
STRPOS(mixed,mixed):smallint	X	X		X	X			X
STRPOS(mixed,dbcs):smallint	X	X		X	X			X
SUBSTR(char,integer,integer):char	X	X	X	X	X	X	X	X
SUBSTR(char,integer):char	X	X	X	X	X	X	X	X
SUBSTR(dbcs,integer):dbcs	X	X		X				X
SUBSTR(dbcs,integer,integer):dbcs	X	X		X				X
SUBSTR(mixed, integer):mixed	X	X		X				X
SUBSTR(mixed,integer,integer):mixed	X	X		X				X
RTRIM(char):char	X	X	X	X	X	X	X	X
RTRIM(dbcs):dbcs	X	X		X				X
RTRIM(mixed):mixed	X	X		X				X
LOOKUP and ERROR set types	X	X	X	X	X	X	X	X
GETRULESHORTNAME	X	X	X	X		X	X	X
GETRULELONGNAME	X	X	X	X		X	X	X
GETRULEIMPNAME	X	X	X	X		X	X	X

[1] INCR has a restriction in AppBuilder 3.2. For details, see *C generation restrictions*.

[2] DECR has a restriction in AppBuilder 3.2. For details, see *C generation restrictions*.

Code Generation Parameters and Settings

Code Generation Parameters and Settings

AppBuilder 3.2 Rules Language Reference Guide

The AppBuilder code generation facility uses many parameters. AppBuilder comes with predefined parameters that are optimized for most applications.

Changing any of the predefined parameters can cause unexpected results.

Parameters that control code generation options can be specified from the command line or in the hps.ini file for the workstation products. For mainframe products parameters can be specified either in the EXEC statement or in the code generation INI file defined by the DD:

//CFG DD DISP=SHR,DSN=&USRVQUAL..CGTABLE(CODEGEN)

where *&USRVQUAL* is the prefix for Enterprise Repository USER datasets that are versioned.

The original member (CODEGEN) within this dataset can be found in &BASEQUAL..CGTABLE where &BASEQUAL is the prefix for Enterprise Repository BASE level datasets. Copy this information into the user dataset above and then modify it.

The following topics are discussed in this chapter:

- INI File Settings
- Command Line Parameters Settings
- Processing Order for Parameters
- Code Generation Limitations
- Supported Codepages

INI File Settings

INI files on the workstation and the Host have similar structures. They are divided into sections. A line starting with semicolon in the first position is the comment line. All the required settings are created by the install; however, you can insert additional settings if necessary. See also Additional Code Generation Settings.
The code generation facility uses the following *language and platform independent* INI file sections:

- [CodeGen]
- [CODEGENPARAMETERS]
- [MacroDomains]
- [MacroDefinitions]
- [CodegenPragmas]

The code generation facility uses the following *language and platform dependent* INI file sections. Settings in these sections apply to only one target language. These settings will overwrite settings from the language and platform independent sections if applicable. See Settings Available in all Language Specific Sections for information about the settings you set in these sections.

Language and Platform Dependent Code Generation Settings

Section	Description
[OpenCobolGen]	This section contains values that overwrite any settings from the common sections and is used only for OpenCOBOL.
[CobolGen]	This section contains values that overwrite any settings from the common sections and is used only for ClassicCOBOL.
[CGen]	This section contains values that overwrite any settings from the common sections and is used only for C.
[CServerGen]	This section contains values that overwrite any settings from the common sections and is used only for C.
[CSharpGen]	This section contains values that overwrite any settings from the common sections and is used only for C#.NET generation, on both client and server side.
[JavaGen]	This section contains values that overwrite any settings from the common sections and is used only for Java.
[JavaServerGen]	This section contains values that overwrite any settings from the common sections and is used only for Java.
[JavaHTMLGen]	This section contains values that overwrite any settings from the common sections and is used only for Java.
[JavaBatchGen]	This section contains values that overwrite any settings from the common sections and is used only for Java.

[CodeGen]

See the following tables for Keys, sample values, and descriptions available in the CodeGen section:

- General settings for the Codegen section
- Workstation specific settings for Codegen section

- Host specific settings for Codegen section

General settings for the Codegen section

This table lists the settings that can be set either on the workstation or on the Host:

General settings for the Codegen section

Key	Possible Values	Descriptions
FLAG	< *flag_name* >	This is used to specify additional code generation parameters or flags. For example, the flag GENNOSUFF can be defined as FLAG=GENNOSUFF. Each flag must be specified on a separate line. The total number of flags cannot exceed 32. See Command Line Parameters Settings for information about setting a flag from the command line.
R2C_CODEGEN_TYPE	C	This setting defines the language and environment. The default is C. B is for COBOL. J is for Java. See -C<parameter_value> parameter for additional values.
R2C_CODEPAGE		This setting defines the codepage for COBOL generation. See Supported Codepages for additional information.
R2C_INCLUDE_DIR	%hpsini%\nt\sys\inc (default)	This setting indicates the directory for include files specified in CG_INCLUDE statement. For more information, see Including Files. If the directory is specified in the code generation parameter, this setting is overwritten. For information about the code generation parameter, see General Parameters.

Workstation specific settings for Codegen section

This table lists the settings for the workstation only.

Workstation specific settings

Key	Possible Values	Descriptions
R2C_MSG_FILE	e:\AppBuilder\ad\cfg\cg\cg.MSG	Location of the code generation error message file
R2C_TABS_DIR	e:\AppBuilder\temp	Location of code generation data files
R2C_CODEGEN_DIR	e:\AppBuilder\ad\cfg\cg	Location of other code generation data files
R2C_BLD_DIR	e:\AppBuilder\ad\cfg\cg	Directory to find temporary BLD files
R2C_WORK_DIR	e:\AppBuilder\temp	Directory to create all temporary work files
R2C_OUTPUT_DIR	e:\AppBuilder\temp	Directory to create output files
R2C_SOURCE_DIR	e:\AppBuilder\temp	Directory to find rule source
R2C_PNL_DIR	e:\AppBuilder\temp	Directory to find window panel files
R2C_BINDFILE_DIR	e:\AppBuilder\bnd	Directory to find rule bind file
R2C_RC_DIR	e:\AppBuilder\temp	Directory to create RC files
R2C_VW_DIR	e:\AppBuilder\temp	Directory to create VW files
R2C_ERR_DIR	e:\AppBuilder\temp	Directory to create temporary error list files
R2C_LST_DIR	e:\AppBuilder\temp	Directory to create listing with rule preparation results

Host specific settings for Codegen section

This table lists the settings for the Host only.

Host specific settings

Key	Possible Values	Descriptions
R2C_MSG_FILE	DD:CFG(CG)	Location of the code generation error message file
R2C_CODEPAGE	Ko_KR.IBM-933	Valid for COBOL and OpenCOBOL only. Code page used for DBCS string functions.

[CODEGENPARAMETERS]

The CODEGENPARAMETERS section contains parameters that control how the code is generated. See the following tables for available settings:

- General settings in CodegenParameters section
- OpenCOBOL specific settings in CodegenParameters section
- Java specific settings in CodegenParameters section
- C specific settings in CodegenParameters section

General settings in CodegenParameters section

This table lists the settings that can be set either on the workstation or on the Host:

General settings in the CODEGENPARAMETERS section

Key	Possible Values	Descriptions
PARAM	< *listparameters* >	This setting is used to list additional code generation parameters.This list is added to the end of the command line parameters. See Processing Order for Parameters for additional information.
DFLTDTFMT	%0m/%0d/%Y	Valid for C, ClassicCOBOL, and OpenCOBOL. This format is used for DATE and CHAR functions when the second parameter is omitted. For Java, see DEFAULT_DATE_FORMAT in appbuilder.ini.
DFLTTMFMT	%0t:%0m:%0s	Valid for C, ClassicCOBOL and OpenCOBOL. This format is used for TIME and CHAR functions when second parameter is omitted. For Java, see DEFAULT_TIME_FORMAT in appbuilder.ini.
DB2DTFMT	%Y-%0m-%0d	Valid for C and ClassicCOBOL only. This format is used to convert date fields when calling DB2.
DB2TMFMT	%0t.%0m.%0s	Valid for C and ClassicCOBOL only. This format is used to convert time fields when calling DB2.
NLSIN	<character_sequence_1>	If both values NLSIN and NLSOUT are defined then they are used to convert all long names from repository or the DCL ENDDCL section. If character_sequence_1 and character_sequence_2 have different lengths then the longer one will be truncated. When the long name is converted, all characters in the name included in NLSIN are replaced with characters from NLSOUT that have the same index. For example, the first character from NLSIN is replaced with the first character in NLSOUT. No verification is done, and the resulting name might be ambiguous.
NLSOUT	<character_sequence_2>	
NLSTABLE	.932	Use NLSTABLE to run code generation with a codepage that is different than the current locale. See C function setlocale for more details on acceptable values. The specified codepage support must be installed on the machine.

CHECK_DEC_FORMAT	YES, NO	This parameter controls constant numeric format strings verification (compile time) for CHAR, INT and DEC functions. Default value is YES. *Note:* Format string validation performed by CHECK_DEC_FORMAT CodeGen parameter is split into runtime validation and compile time validation: • Runtime validation is controlled by CHECK_DEC_FORMAT parameter from [AE RUNTIME] in the HPS.ini section for generation that uses C runtime. The default value is NO. • Runtime validation for Java is controlled by the CHECK_DEC_FORMAT parameter from the [NC] section of APPBUILDER.ini. Runtime default value is NO. For more details, see also Format string validation. For details about error handling issues due to new format string validation rules, see Format String Validation Error Handling.

OpenCOBOL specific settings in CodegenParameters section

This table lists the settings that are supported for OpenCOBOL only.

OpenCOBOL specific settings in the CODEGENPARAMETERS section

Key	Possible Values	Descriptions
CompareDatesAsString	N	This setting controls how date comparisons are generated. If set to Y, date fields are compared as strings where the result is defined by the environment. If set to N, date fields are converted to integer values and compared. This comparison is platform independent. In AB2032 string comparison is used only in host variables. See DATEDB2CMP. In all other cases, the date values are converted to integer values and then compared.
DATEDB2CMP	0001-01-01	This is the value to compare all host Date values with. See DATEDB2DFLT for more details.
DATEDB2DFLT	0001-01-01	The value to assign to a DATE host variable if the variable is less than DATEDB2CMP.
DATEFMT	%Y-%0m-%0d	This format is used when a support library function is called. Only the delimiter can be changed by the end user.
DATEINIT	0000-00-00	This setting is used to initialize DATE type fields. Only delimiters and separate digits can be changed by the end user, however structure should remain the same. Note that corresponding delimiters in DATEFMT and DATEINIT parameters should coincide.
DEFAULT_CENTURY	1900	This is the value for the century in the DATE function when only two digits are used for the year in the input and format strings. This value would be added to the two digit year when parsed. This value is also passed to the support library functions. The default value is 1900. To change this value for runtime, the rule must be re-prepared. This setting can be overridden by PRAGMA CENTURY statement. See also DEFAULT_CENTURY in appbuilder.ini and DEFAULT_CENTURY in the [AE Runtime] section of the HPS.INI.
DFLTTSFMT	%0o-%0d-%Y.%0t.%0m.%0s.%f	This format is used for TIMESTAMP and CHAR functions, when the second parameter was omitted.

OCC_VIEW_SIZE_THRESHOLD	<integer>	This setting specifies the threshold of the occurring view size to determine whether or not fields in the occurring views are initialized. This setting must be used with the code generation parameter flag -FNCOCC. For example, if OCC_VIEW_SIZE_THRESHOLD=299, all occurring views are initialized if the number of occurrences is less or equal to 299, but they are NOT initialized if the number of occurrences is 300 or greater. If this setting is zero or not specified, all occurring views are initialized.
SQLINITFLAG	0	This controls whether SQL-INIT-FLAG is reset to 0 in each rule. If the setting is equal to 0, the MOVE ZERO TO SQL-INIT-FLAG is generated for each database rule. By default, SQL-INIT-FLAG is not initialized. *Note:* When COBOL SQL co-processor for DB2 is used, this flag should be removed (disabled).
TIMEFMT	%0t.%0m.%0s.%0f	This format is used when a support library function is called. Only the delimiter can be changed by the end user.
TIMEINIT	00.00.00.000	This value is used to initialize TIME type fields.
TIMESTAMPDB2CMP	0001-01-01-00.00.00.000000	The value to compare all host TIMESTAMP variables with. See TIMESTAMPDB2DFLT for more details.
TIMESTAMPDB2DFLT	0001-01-01-00.00.00.000000	The value to assign to a TIMESTAMP host variable if it is less than TIMESTAMPDB2CMP.
TIMESTAMPFMT	%Y-%0o-%0d-%0t.%0m.%0s.%0f	This format is used when support library function is called. Only delimiters can be changed by the end user.
TIMESTAMPINIT	0000-00-00-00.00.00.000000	This value is used to initialize TIMESTAMP type fields.

Java specific settings in CodegenParameters section

This table lists the settings that are supported only for Java.

Java specific settings in the CODEGENPARAMETERS section

Key	Possible Values	Descriptions
SYSTEMVIEWPACKAGE	appbuilder.systemviews	This is not available in AB3.1 or later versions.
GLOBAL_PACKAGE		This is not available in AB3.1 or later versions.
SET_PACKAGE	set	This is not available in AB3.1 or later versions.
REFLECTION_VIEW_MAP		This is not available in AB3.1 or later versions.
STATIC_CLEAR		This is not available in AB3.1 or later versions.
VIEW_PACKAGE	view	This is not available in AB3.1 or later versions.
COMPONENT_PACKAGE	component	This is not available in AB3.1 or later versions.
COPYFROM_NULL_PARAMETERS_THRESHOLD	[number(percent)]	This number is a threshold percentage for determining a way of views mapping generation. The first option is to use view copyFrom method with a list of view fields. It is used when the percentage of destination view fields, which have no corresponding field in the source view (i. e. passed as null to copyFrom method), is less then this parameter value. Otherwise the second option is: field-to-field assignment is generated directly in the rule. Set this parameter to 100 for best generated code maintainability. Set it to 0 for best performance. Default value is 10.

GENERATE_RULE_CALLS_TRACE		When this is set to YES, debug level TRACE statements are generated at the beginning and end of each rule. When this is set to NO, debug level TRACE statements are not generated. Debug level TRACE statements are generated conditionally, so these statements can be disabled at runtime, if the debug option is turned off.
AUTOCOMMIT	false	By default, DB2 creates a connection with AutoCommit set to TRUE, which disables all transaction control. To enable transaction control with SQL statements, COMMIT and ROLLBACK connection method setAutoCommit(false) must be invoked. Generated Java code will have setAutoCommit() call with a parameter equal to the value specified by AUTOCOMMIT (case-sensitive). If the AUTOCOMMIT key is not set, no call is generated.
JAVA_PERSISTENT_CURSOR	[YES][NO]	This parameter controls SQL cursor persistence. If set to YES then an SQL cursor will be persistent. Persistence means that a cursor created by a particular rule must be retained past the end of the rule invocation and made available to subsequent invocations of that rule within the same scope until explicitly closed; however, in the case of executing in a server request scope, the request terminates. Setting to YES is equivalent to setting command line flag SQLPERSIST. Possible values are YES or NO (default).
READ_ONLY_SQL_CURSOR	[YES][NO]	This parameter configures the generation of an SQL cursor which has no explicit FOR UPDATE specification. When this parameter is set to YES, an SQL cursor having no explicit FOR UPDATE specification is generated as read-only sqlj iterator. If there is an attempt to perform an UPDATE or DELETE operation positioned by this cursor in the rule code, the code generation submits an error. An SQL cursor which has an explicit FOR UPDATE clause in its declaration is generated as an updatable sqlj iterator. When this parameter is set to NO, an SQL cursor having no explicit FOR READ ONLY (FETCH ONLY) specification is generated as an updatable sqlj iterator. Cursor explicitly declared as READ ONLY is generated as a read-only iterator.
FLOATING_POINT_STANDARD	IEEE754 HEXADECIMAL	This parameter specifies the floating point types ranges.
DATA_CONVERTER	<qualified java class name>	This parameter specifies the Java class implementing the appbuilder.util.AbfDataConverter interface, a class used for converting data (views, fields) to a char[] array and reverse. The data conversion is performed for OVERLAY and REDEFINE operations. Also, data converter counts the size of data as a size of its converted representation. The result of size count is the result of Rules SIZEOF std function.
ASSERT_VIEW_IDENTITY	[YES][NO]	Default value is YES. YES or NO indicates whether CodeGen should generate assertIdentity calls to ensure that the views used in the rule are of the proper version. ASSERT_VIEW_IDENTITY also specifies whether to generate getHash() method in a view class. If set to YES, then the getHash method is generated and called to ensure the proper view version. If set to NO, then the getHash method is not generated and not called.
GENERATE_VIEW_FIELD_ACCESSORS	[YES][NO]	This flag is valid for Java generation only. Default value is NO. If set to YES, the generated view java class contains the view field accessor methods (getXXX setXXX). If set to NO, the public field accessors are not generated.

GENERATE_IO_VIEW_TRACE	[YES][NO]	This setting affects Java generation only. When GENERATE_IO_VIEW_TRACE is set to YES the input view trace is generated at the beginning of the rule, while the output view trace is generated at the end of the rule. TRACE statements are generated conditionally, so these statements can be disabled at runtime. At runtime they are controlled by APP_LOGGER.INFO level.
ALWAYS_GENERATE_USER_TRACE	[YES][NO]	This setting affects Java generation only. With ALWAYS_GENERATE_USER_TRACE codegen parameter set to YES, all user-coded TRACE statements are always generated in the target code (even with Rule debug option off) and can be enabled or disabled through runtime settings. When set to NO and codegen command-line flag GENTRACE is specified, no user TRACE statements are generated in the Java code. When debugging info generation is ON (-yd commandline parameter is not specified for codegen), all user TRACE statements from the rule source code are generated.
GENERATE_STATELESS_RULE	[YES][NO]	This setting affects Java generation only. When set to YES, it determines the rule stateless generation, when possible. The default value is NO.
INLINE_VIEW_COPY	[YES][NO]	When set to YES, the sequence of Java statements corresponding to rules MAP statement with view arguments is generated in the rule class as is (inline). When set to NO, this sequence is enclosed to a private rule class method and MAP is generated as call to the method (see also INLINE_VIEW_COPY_FIEL DS_LIMIT). The default value is NO.
INLINE_VIEW_COPY_FIEL DS_LIMIT	<number>	When INLINE_VIEW_COPY is set to NO, this number determines the maximum number of fields in a view V for which statement MAP ... TO V is generated inline. If the number of fields exceeds this limit, then the Java code corresponding to MAP with destination view V is enclosed in a private rule class method (see also INLINE_VIEW_COPY). The default value is 2.
EXPAND_RULE_SIGNATURE	[YES][NO]	This setting affects Java generation only. When set to YES, the rule and subrule signatures are expanded if possible. The default value is NO.
MAX_SUMMARY_IO_VIEW_FIELDS	255	This setting affects Java generation only. It sets the maximum number of input and output view fields (summary) for which expanding is provided.
LAZY_INSTANTIATION_ENABLED	[YES][NO]	If this parameter is set to YES, all local variables and views (excluding redefined views and input/output views) are instantiated when they are explicitly accessed for the first time and released after the last use. The default value is NO.

C specific settings in CodegenParameters section

This table lists the settings that are supported only for C.

C specific settings in the CODEGENPARAMETERS section

Key	Possible Values	Descriptions
INDEX_CONTROL_ON	YES	Specify if generated C code should perform index checking when accessing occurring views. Values: YES or NO
INDEX_CONTROL_ABORT	NO	Specify if rule should abort at execution time if index is out of bounds when accessing occurring views. Values: YES or NO

[MacroDomains]

This section contains Macro Domains definitions. See Validating Macros in Domain for more information.

Settings in the MacroDomains section

Key	Possible Values	Descriptions
LANGUAGE	Java,C,COBOL,OpenCOBOL	This setting defines all possible values for macro LANGUAGE.
ENVIRONMENT	Server,HTML,GUI	This setting defines all possible values for macro ENVIRONMENT.

[MacroDefinitions]

This sections defines macros that can be used for all target languages and platforms. The MacroDefinitions section can be viewed and updated from *Construction Workbench* > *Tools* > *Workbench Options* > *Preparation tab* > *Conditionals* button.

[CodegenPragmas]

PRAGMA statements are special commands that control certain features of the compiler.

```
<pragma_line>

...
<pragma_line>
```

where pragma_line has correct Rules Language syntax. See Compiler Pragmatic Statements for more information.

Settings Available in all Language Specific Sections

The following table lists key settings, sample values, and descriptions of parameters that can be used in the language and platform dependent sections listed in Language and Platform Dependent Code Generation Settings:

Settings available in all language specific code generation sections

Key	Possible Values	Descriptions
MACRO	LANGUAGE=Java ENVIROMENT=GUI	This defines a macro that is language or environment specific.
PARM	PARAM=< *listparameters*>	This setting is manually inserted by the user, and overwrites the PARAM setting from the [CodegenParameters] section.
PARM	<ParameterName>=<parameter_value>	This setting is manually inserted by the user, and overwrites the setting from the [CodegenParameters] section with name ParameterName giving it the value parameter_value.
R2C_STANDARD_TABS	e:\AppBuilder\ad\cfg\cg\javas.tab	Specifies the location of the code generator data file.
R2C_OUT_EXT	.java	Specifies the extension to use for all generated programs

Examples of settings in language specific sections

```
[JavaGen]
R2C_STANDARD_TABS=e:\AppBuilder\ad\cfg\cg\javas.tab
; location of code generator data file
R2C_OUT_EXT=.java
; extension to use for all generated programs
MACRO=LANGUAGE=Java
MACRO=ENVIRONMENT=GUI
; define a macro that is language or environment specific

[COBOLGEN]
R2C_STANDARD_TABS=DD:CFG(COBOLTAB)
; location of code generator data file

PARM=PARAM= -H -VMO -!O -J -YG
; Default parameters setting for COBOL generation

[OPENCOBOLGEN]
R2C_STANDARD_TABS=DD:CFG(OCOBOLT)

PARM=PARAM= -VMC -fdyncall -yz
; Default parameters setting for Open COBOL generation
PARM=DATEINIT=0000/00/00
```

Additional Code Generation Settings

The DECIMAL_ARITHMETIC_MODE setting in the [AP Windows] section of the Hps.ini file defines the arithmetic used in the generated program. Three values are supported:

- COMPATIBILITY for HPS532 compatibility
- COBOL for COBOL compatibility
- CALCULATOR uses calculator rules and is compatible HPS540 NT. This value must be set using the configurator.

Command Line Parameters Settings

You can set additional code generation parameters from the command line. You can also list these parameters as values for the PARAM settings in the INI file. Each parameter must begin with the dash symbol (-) . Parameters must be separated by at least one space. Parameters are not case sensitive, with the exception of -P. See Processing Order for Parameters for information about how the INI settings and command line parameters are processed.

- General Parameters
- Workstation Specific Parameters
- ClassicCOBOL and OpenCOBOL Specific Parameters
- View Initialization Parameters
- ClassicCOBOL Specific Parameters
- OpenCOBOL Specific Parameters
- C Specific Parameters
- Java Generation Parameters

General Parameters

If restrictions are not specifically mentioned in the parameter description, the parameter can be used for all platforms, languages, and environments.

- -C<parameter_value>
- -F<flag_name>
- -H
- -J
- -P<parm_name=value>
- -yk
- -V<parameter_value>
- -yp[<pragma_line>]
- -yx<macro_name=macro_value>
- -DU<dir>
- -ym
- -yt

-C<parameter_value>

-C parameter defines the language and environment as follows:

C	*Generates a C program. The second letter defines environment as follows:*	
	S	server
	C	client. **This is the default value.**
B	*Generates a ClassicCOBOL program.*	
J	*Generates for Java generation. The second letter defines environment as follows:*	
	S	server
	C	client. **This is the default value.**
	H	HTML / Servlet
	B	Java Batch
O	*Generates for OpenCOBOL. The second letter defines environment for user calls as follows:*	
	Y	Passes the parameters DFHEIBLK and HPSCOMMAREA
	N	Passes only the input and output views as parameters. **This is the default value.**
	B	Passes dummy DFHEIBLK and HPSCOMMAREA along with the input and output views.
	C	Passes DFHEIBLK and HPSCOMMAREA along with the input and output views.
S	*Generates for C# .NET. The second letter defines environment as follows:*	
	C	client. **This is the default value.**
	S	server
	B	batch (nearly the same as server but standalone)
	W	WPF client

In the following example, *-C* is the parameter for selecting language and environment, and *B* specifies that ClassicCOBOL is generated:

-CB

-F<flag_name>

-F parameter is used to specify additional code generation parameters referred to as flags. For example, the flag MEXCI can be passed to code generator as -FMEXCI. See also FLAG for setting a flag in the INI file.
The following flag_name can be specified in the parameter list using -F<flag_name>:

Flag Name	Description
CMNT	Include user's comments from the rule in the generated code.
NOSRC	Do not include rule source lines in the generated code.
NOLINE	Do not include rule source line number information in the generated code. This option is not supported for C.
MEXCI	Use case-insensitive comparison in macros. See Case-sensitivity for details.

MEXPDFMOFF	Switches off the following predefined macros: • CG_RULE_TRANSLATION_DATE • CG_RULE_TRANSLATION_TIME • CG_RULE_TRANSLATION_TIMESTAMP • CG_CODEGEN_VERSION • CG_RULE_SHORT_NAME • CG_RULE_LONG_NAME • CG_RULE_IMP_NAME • CG_DEBUG For more information about the predefined macros, see Predefined Macros.
PNC	Normally, code generator verifies that a rule calls other rules and components for the supported platforms only. With this flag there is no verification. In this case it is the developers' responsibility to ensure that generated code is correct.
URCOPT	Disables optimization for unreachable code. This is not supported for ClassicCOBOL or OpenCOBOL.
SHORTINTRO	It controls the generation of extended information at the beginning of every generated file. • When this flag is *not specified* , cogeden produces common information about the list of changes. • When this flag is *specified* , codegen produces a short description of the date and time when the code was generated, also offering details about the codgen's version. For Java, when this flag is not specified, the information produced by codegen looks as follows: /* Code generated by: AppBuilder * rule: CDGN_OBJ_REF_1_R * Code generated on: Mon Jan 10 15:33:51 2005 * Codegen version CG0100_RFX. Build Dec 16 2004 at 17:04:06 * Options used: -cjc -vdn -ye -ym -j -dfe:\ad\codegen\codegen.cfg * -fparms * [CODEGENPARAMETERS] * DFLTDTFMT=%0m/%0d/%Y * DFLTTMFMT=%0t:%0m:%0s * DB2DTFMT=%Y-%0m-%0d * DB2TMFMT=%0t:%0m:%0s * SYSTEMVIEWPACKAGE=com.level8.appbuilder.systemviews * GLOBAL_PACKAGE= * SET_PACKAGE= * VIEW_PACKAGE= * COMPONENT_PACKAGE=component * OCC_VIEW_SIZE_THRESHOLD=1000 */ When this flag is specified, codegen will produce short description: /* Code generated by: AppBuilder * rule: GIFT_SETVIEW_SL_DIS * Code generated on: Thu Apr 07 05:37:01 2005 * Codegen version CG0101_BASE:CG0101_RFX. Build Apr 6 2005 at 14:18:45 */
NOMIXVAL	Use this flag to avoid the generation of DBCS and MIXED string validation for Classic COBOL code. When used, the generated code does not check the validity of DBCS and MIXED strings for validity, clean invalid ones and invoke DBCS-ABEND handler.

SIZEOF	This flag helps you to calculate the SizeOf function call as *sizeof (one_view_occurrence)*number_of_occurrence* . If this flag is not used, then SizeOf is calculated as *sizeof (one_view_occurrence)* . When applied to a view with more than one occurrence, the SIZEOF function produces different results in C than on other platforms, like in the example below, where, for the view V: *DCL* *i integer;* *v view contains i;* *vv view contains v(10);* *ENDDCL* the SIZEOF function returns 4 (size of the INTEGER field) in ClassicCobol, OpenCobol and in Java modes, while in C it returns 40, which is 4 multiplied by the number of occurrences, i.e. 10. Without the SIZEOF flag, the SIZEOF of a view is NOT multiplied by the number of occurrences (in the previous example, the result will be 4 in C mode too); with this flag specified, the result is always multiplied by the number of occurrences and is 40 for all the platforms.

-H

-H parameter is deprecated for ClassicCOBOL generation. It may be used but it has no effect on the generated code. Starting with AppBuilder 3.1 and RFX CG0304 only necessary conversions are performed. It is possible, but highly unlikely that the default standard parameters for ClassicCOBOL generation were modified to exclude -H and to take advantage of the incorrect values for some host variables. If this is the case then the rule code should be reviewed to ensure that it is still works the accepted way. For details about -H parameter, see Using SQL host variables in the rules for Classic COBOL, OpenCOBOL, and C generation.

-J

-J parameter enables DBCS support.

-P<parm_name=value>

This parameter has the following restrictions:

- The parm_name cannot contain spaces or equal (=) symbols.
- The value starts with the first symbol after equal (=) sign and ends with the first space.
- The value after equal (=) sign is case-sensitive.

-yk

Use this setting to disable new keywords. See Reserved Words.

-V<parameter_value>

-V parameter controls how some constructs are generated as follows:

M	*Defines how to generate and evaluate math statements.*	
	N	Use compile time optimization and CALCULATOR mode. *Note:* CALCULATOR arithmetics is used as default for Java and C generations.
	O	Compatible with HPS 5.3, and compile time optimization is not performed. *Note:* This is the default value.

	C	Use COBOL rules. *Note:* (COBOL arithmetic) is used by default in the cases of ClassicCOBOL and OpenCOBOL generations
T	*Defines how to generate subtraction INT - TIME.*	
	N	Produces TIME value (DEFAULT).
	O	Produces INT value.
C	*Defines how to format ClassicCOBOL code.*	
	L	Uses long name based identifiers.
	S	Uses short/implementation name based identifiers. This is the default.

-yp[<pragma_line>]

The < *pragma line* > must specify pragma construction with Rules Language syntax. For example:

-yp[pragma keywords off (object)]

-yx<macro_name=macro_value>

This parameter defines macro with the name specified by *macro_name* and the value specified by *macro_value* . The following restrictions apply:

- The *macro_name* and *macro_value* cannot contain spaces or the equals (=) symbol.
- The *parm_name* and *macro_value* after the equal (=) sign are case-sensitive.

For example,

-YXTARGET=Java

Macro definitions can also be set in the ini files. See [MacroDefinitions] for more details.

-DU<dir>

Specifies the directory for include files. -DU setting overwrites the R2C_INCLUDE_DIR setting in the ini file.
For example, the following parameter specifies the directory to be DD:INCLUDE.

-DUDD:INCLUDE

-ym

Specifies whether macro preprocessor is invoked by the code generator.

-yt

The code generation command line parameter -yt can be used to specify a database engine:

-yt<db>

where <*db*> is one of:

- *db2* – for IBM DB2
- *ora* – for Oracle
- *sql* – for Microsoft SQL Server.

When using the Oracle option, the "FOR READ ONLY" clause of cursor declaration is skipped in result Java code.

Workstation Specific Parameters

Unless otherwise noted in the parameter description, settings can be used for all supported languages and environments.

- -YC<parameter_value>
- -F<flag_name> for Workstation
- -E<parameter_value><Ext>
- -D<parameter_value><File/DirName>
- -N<parameter_value>
- -G<country>

-YC<parameter_value>

This parameter is used only for client side OpenCOBOL. It sets the language for generated view, and specifies copybooks used during component and set preparation. The following table shows the possible values for this parameter:

C
PL/I
COBOL
OPENCOBOL
Assembler

-F<flag_name> for Workstation

The following flag_name can be specified in the parameter list using -F<flag_name>:

Flag Name	Description
OCDIR	This option is used only for client-side OpenCOBOL. If the option is specified, then views are generated in <R2C_OUTPUT_DIR>\view and sets are generated in <R2C_OUTPUT_DIR>\set where <R2C_OUTPUT_DIR> is the value from HPS.INI. If the rule has no views or sets attached, then it is not necessary for either directory to exist, or they can be empty. If this option is not specified, views and sets are generated to <R2C_OUTPUT_DIR>.
SCCOPY	This option is used only for client-side OpenCOBOL. If this option is specified, only copybooks for sets and views are generated. Generated copybooks language is determined by the -YC parameter.

-E<parameter_value><Ext>

-E parameter defines the extension for files types as specified by the following second letters. The < *Ext* > must include the leading period (.). The following table describes the possible values for this parameter:

I	LOG file extension
X	CGMEX file extension
B	bind file extension
G	output bind file extension
R	rule file extension
O	generated C or COBOL file extension
C	RC file extension
W	VW file extension
E	errors file extension
L	listing file extension

T	tables file extension
P	panel file extension
Q	tree file extension
D	BLD file extension
H	Panel script file extension
K	RW script file extension
SVO	OpenCOBOL View copybook
SSO	OpenCOBOL Set copybook
SVB	Component View COBOL copybook
SVP	Component View PL\1 copybook
SSB	Set COBOL copybook
SSP	Set PL\1 copybook
SSC	Set C copybook
SSA	Set Assembler Source

For example, the following parameter sets extension for generated C code to 'C' instead of 'PCC', which is the default:

-EC.C

-D<parameter_value><File/DirName>

-D parameter defines the directory or file name for the following files:

B	Directory that contains bind file
C	RC file directory
D	BLD file directory
E	Errors file directory
F	Config file name
G	Code Generation home directory
I	LOG file directory, DEFAULT WorkDir
K	Work directory for temporary files
L	Listing file directory
M	Messages file name
N	Standard code generation tables name
O	Directory that contains generated C or COBOL program
P	Directory that contains panel file
Q	Tree files directory
S	Directory that contains source rule file
T	Tables file name
W	VW file directory
X	Directory that contains generated CGMEX file, DEFAULT WorkDir
H	Panel script file directory

R	RW script file directory

-N<parameter_value>

-N parameter specifies the name of the file to use during code generation as follows:

X	CGMEX file name
I	LOG file name
Q	Tree file name
T	Tables file name
E	Error file name
D	BLD file name
B	Bind file name
P	Panel file name
H	Panel script file name
R	RW script file name

-G<country>

This parameter has the same functionality as NLSTABLE in the INI file.

ClassicCOBOL and OpenCOBOL Specific Parameters

These parameters are only supported for ClassicCOBOL and OpenCOBOL.

- -YZ
- -yi

-YZ

In OpenCOBOL, this disables DATE and TIMESTAMP verification, which normally occurs before they are passed to SQL statements.
In ClassicCOBOL, this disables generating IF (... < 367), which verifies that the date value is correct when converting an AppBuilder date to an SQL date and back.
For more details, see the description of ini file settings DATEDB2CMP and TIMESTAMPDB2CMP.

-yi

This parameter uses the INITIALIZE statement to initialize or CLEAR views. The generated code uses the INITIALIZE statement to initialize views instead of the MOVE SPACES/ MOVE ZEROES method. View Initialization Parameters also affect how views are initialized.

View Initialization Parameters

The parameters in this section are supported only for ClassicCOBOL and OpenCOBOL unless otherwise noted. The default parameters provide a logically correct application; however, it might not be the best performance alternative. AppBuilder provides several optional code generation parameters. Analyze and test your application to determine which parameters work best for your application. For example, you can choose between the INITIALIZE statement and the older MOVE SPACES / MOVE ZEROES method.
There are also several options that use statically-initialized structures in COBOL working storage that map to the identical structure. Determine which is best for your installation. The MOVE SPACES / MOVE ZEROES default might be more efficient if your views have an OCCURS clause because the INITIALIZE view statement generates more assembler statements in the COBOL II compiler.
Views available for a rule fall into two categories:

- *Views visible only for a given rule:* These are local variables, auxiliary and temporary variables used in generated views and BINDFILE views declared as Working views. These views are put into the WORKING-STORAGE section during code generation.
- *External views, visible to other rules and components:* These are rules and components, input/output views, and global views. These views are put into the LINKAGE section during code generation.

All WORKING-STORAGE and some LINKAGE section views are required to be initialized at the beginning of rule execution.

If specified, AppBuilder can create a duplicate copy of the view in COBOL working storage, then move the COPY structure to the view during initialization. Depending on view sizes, occurring views, and number of numeric fields, this can improve performance significantly. However, it also increases the size of the load module by the size of the views. This might require maintenance if the application must be moved because of disk space considerations.

-F<flag_name> for View Initialization

The following flag_name can be specified in the parameter list using -F<flag_name>. Only one flag can be used at a time:

Flag Name	Description
LVI	This parameter initializes the linkage view statically. A view copy is created in working storage and initialized statically. The "MOVE copy TO view" statement is used each time the view is initialized or the CLEAR statement is used. In this case, the -YI parameter is ignored.
WVI	This parameter initializes the working storage view statically. A view copy is created in working storage and initialized statically. The "MOVE copy TO view" statement is used each time the view is initialized. In this case, the -YI parameter is ignored.
IP	This parameter generates the INITIAL property. The INITIAL property places a program and any programs it contains in their initial states. A program is in its initial state when: • Data items having VALUE clauses are set to the specified value. • Altered GOTOs and PERFORM statements are set to their initial states and internal files are closed.
SWVI	This is not supported in OpenCOBOL. This parameter initializes working storage view statically using the VALUE clause. The rule code must take care of its local and working view initialization; otherwise, a subsequent rule call might lead to an error. When the CLEAR statement is used, the view is initialized using the INITIALIZE statement. In this case, the -YI parameter is ignored.

ClassicCOBOL Specific Parameters

These parameters are only supported for ClassicCOBOL.

- -YG
- -KNN
- -yd
- -YO and -!O

-YG

This parameter enables global view support.

-KNN

All expressions with decimal fields with lengths less than NN are generated using native COBOL arithmetic. NN must be a number between 15 and 31. The default value is 18.

-yd

When this parameter is set, the GOTO DEPENDING ON statement in the beginning of a COBOL rule is not generated.

-YO and -!O

Setting either of these parameters enables generation of additional parameters for HCGOPER call.
-VMO is required for -!O.

OpenCOBOL Specific Parameters

The following parameters affect the style of the generated code:

- -YQM
- -YQF
- -YQU
- -F<flag_name> for OpenCOBOL

-YQM

When this parameter is set, the minimum number of qualifications for the fields are generated in the view.

-YQF

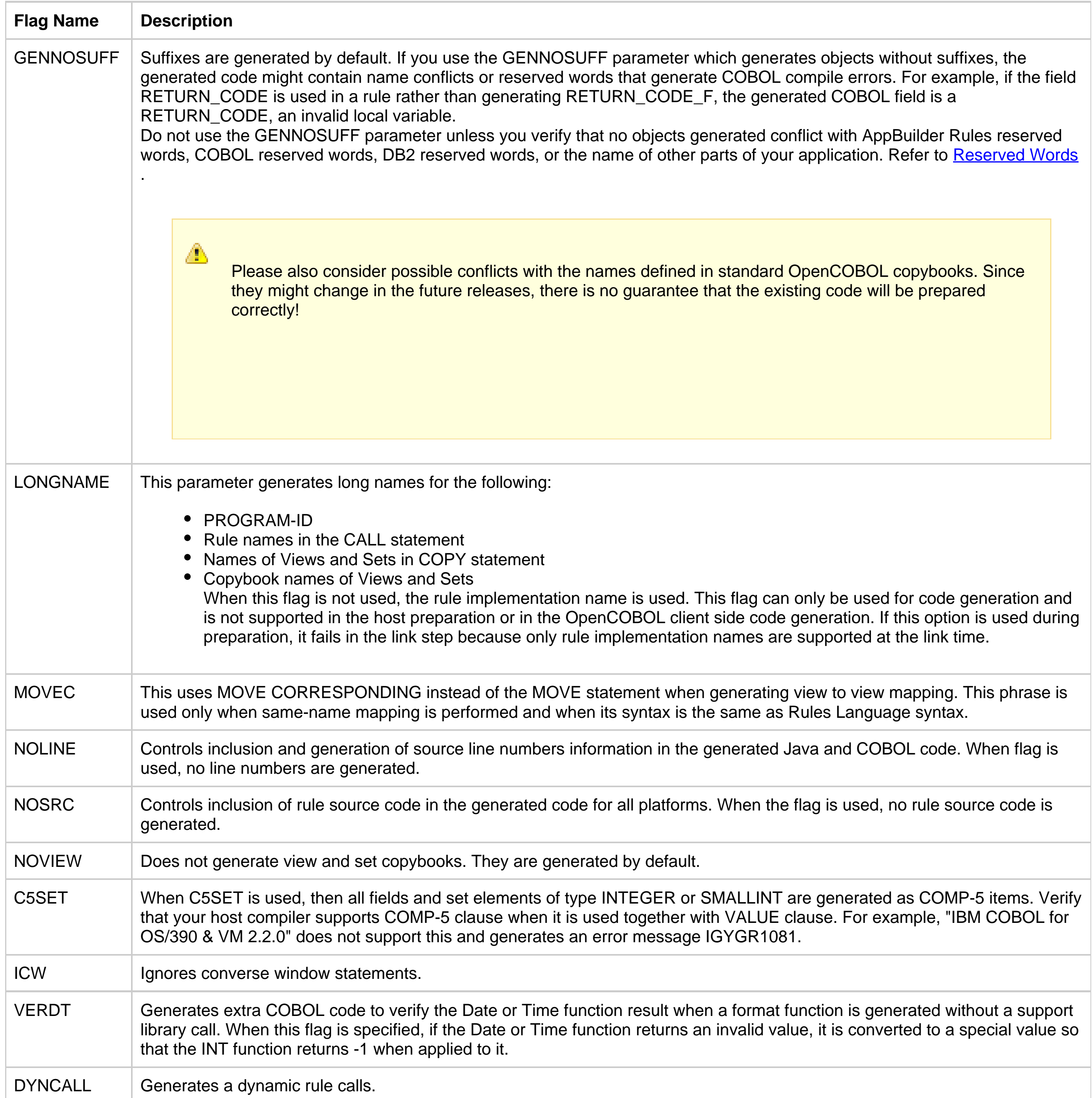
When this parameter is set, the full qualification for the fields is generated in the view. This is the opposite of YQM.

-YQU

When this parameter is set, only those qualifications that were used in the rule source are generated. This is the default.

-F<flag_name> for OpenCOBOL

The following flag_name can be specified in the parameter list using -F<flag_name>:



Flag Name	Description
GENNOSUFF	Suffixes are generated by default. If you use the GENNOSUFF parameter which generates objects without suffixes, the generated code might contain name conflicts or reserved words that generate COBOL compile errors. For example, if the field RETURN_CODE is used in a rule rather than generating RETURN_CODE_F, the generated COBOL field is a RETURN_CODE, an invalid local variable. Do not use the GENNOSUFF parameter unless you verify that no objects generated conflict with AppBuilder Rules reserved words, COBOL reserved words, DB2 reserved words, or the name of other parts of your application. Refer to Reserved Words . ⚠ Please also consider possible conflicts with the names defined in standard OpenCOBOL copybooks. Since they might change in the future releases, there is no guarantee that the existing code will be prepared correctly!
LONGNAME	This parameter generates long names for the following: • PROGRAM-ID • Rule names in the CALL statement • Names of Views and Sets in COPY statement • Copybook names of Views and Sets When this flag is not used, the rule implementation name is used. This flag can only be used for code generation and is not supported in the host preparation or in the OpenCOBOL client side code generation. If this option is used during preparation, it fails in the link step because only rule implementation names are supported at the link time.
MOVEC	This uses MOVE CORRESPONDING instead of the MOVE statement when generating view to view mapping. This phrase is used only when same-name mapping is performed and when its syntax is the same as Rules Language syntax.
NOLINE	Controls inclusion and generation of source line numbers information in the generated Java and COBOL code. When flag is used, no line numbers are generated.
NOSRC	Controls inclusion of rule source code in the generated code for all platforms. When the flag is used, no rule source code is generated.
NOVIEW	Does not generate view and set copybooks. They are generated by default.
C5SET	When C5SET is used, then all fields and set elements of type INTEGER or SMALLINT are generated as COMP-5 items. Verify that your host compiler supports COMP-5 clause when it is used together with VALUE clause. For example, "IBM COBOL for OS/390 & VM 2.2.0" does not support this and generates an error message IGYGR1081.
ICW	Ignores converse window statements.
VERDT	Generates extra COBOL code to verify the Date or Time function result when a format function is generated without a support library call. When this flag is specified, if the Date or Time function returns an invalid value, it is converted to a special value so that the INT function returns -1 when applied to it.
DYNCALL	Generates a dynamic rule calls.

RTCALL	Generates all Date/Time/Char conversion functions as library calls.
NOVIEW	Do not generate copybooks for views and sets. They are generated by default.
GENPERIOD	Generate period after each COBOL statement.
NOSSR	Generate more efficient and readable COBOL code without support for COBOL compiler SSRANGE option. NOSSR is the default. If NOSSR is not specified, the generated code for all rules statements with exception of TRACE is safe to be used with SSRANGE COBOL compiler option. String concatenation and some other constructions are less efficient in this case. The main source for incompatibility with the COBOL compiler SSRANGE option is empty VARCHAR fields that have length 0, but 0 is not allowed in the reference modification for COBOL.
RTDTI	Generate INT(date) function as library call. With support library call, INT(date) function performance is approximately twice better comparing to a native COBOL version.
UNIX	This flag must be specified when generating OpenCOBOL for HP-UX. When UNIX is specified: • SOURCE-COMPUTER and OBJECT-COMPUTER phrases are not generated. • Calls to the following system components are generated directly, for example, not using SYSCOMP proxy, DFHEIBLK and RULE-COMP-COMMAREA: - HPS_OPEN_FILE_LOCATE_MODE - HPS_CLOSE_FILE_LOCATE_MODE - HPS_WRITE_FILE_LOCATE_MODE - HPS_READ_FILE_LOCATE_MODE - HPS_TRUNC_FILE_LOCATE_MODE - HPSPARM • Call to HPSMODE component returns "BATCH". • Call to HPS_GET_ENVIRONMENT returns "UNIX". • Call to TIMESTAMP function without arguments is generated differently. Precision of fraction of returned TIMESTAMP value is 6 digits in this case. • If rule contains calls to system components, but not to user components, HPSCOMM copybook is not generated. • SQLCA copybook renamed to ABSQLCA. • EXEC SQL INLCUDE is used instead of COPY to include view copybooks. Note. Unix prepare also requires flags C5SET and RTDTI.
AIX	This flag must be used instead of UNIX flag when target platform is AIX. *Only one flag AIX or UNIX is allowed!*
NCOCC	Do not initialize occurring views. This flag must be used with the initialization setting OCC_VIEW_SIZE_THRESHOLD in the [CODEGENPARAMETERS] section. For more information, see Initialization of Occurring Views in OpenCOBOL.
OPTNOCOND	In case when conditional statement has empty then-part, it is optimized usually in the following way: if <condition> then else <statements> endif is converted to if not <condition> then <statements> endif The OPTNOCOND flag is used to avoid this optimization.
UC	This flag must be used when the targtet platform is AIX running with Korean codepage.
NOINSPSTRP	This flag controls the generation of the STRPOS function call. When this flag is specified, the INSPECT statement is not used for the STRPOS function call. *This flag is supported for OpenCOBOL only!*
NATIONAL	This flag enables a special implementation of Character String Functions for the strings containing national symbols. This parameter generates all calls of UPPER and LOWER functions so that national symbols in the argument string are correctly handled. When this flag is specified, functions UPPER and LOWER are implemented (using the temporary variable with NATIONAL data type which receives the argument of the function) and are passed to COBOL functions UPPER-CASE or LOWER-CASE. The CODEPAGE(N) parameter with the corresponding codepage number for the Host is passed to the COBOL compiler. For OpenCOBOL, this parameter might be added to the OCCPARM1= or OCCPARM2= values in the section (COBOL) of @MSVENV initialization member.
VCTRACE	This flag instructs codegen to generate an IF condition for every VARCHAR argument of TRACE statement to verify that its length is not zero. If it is not specified, clearing of the varchar variable directly before tracing might lead to an abend at run-time.

C Specific Parameters

The following parameter is supported for C generation only.

-I

This parameter disables the generation of index checking for subscripted views.
If this parameter is specified, no index checking routines are generated. If subscript is out of range, the application might or might not abort; in either case, the first occurrence is not assumed.
This parameter also disables effect of INDEX_CONTROL_ON and INDEX_CONTROL_ABORT Hps.ini settings.

Java Generation Parameters

The following parameters are supported for Java generation only.

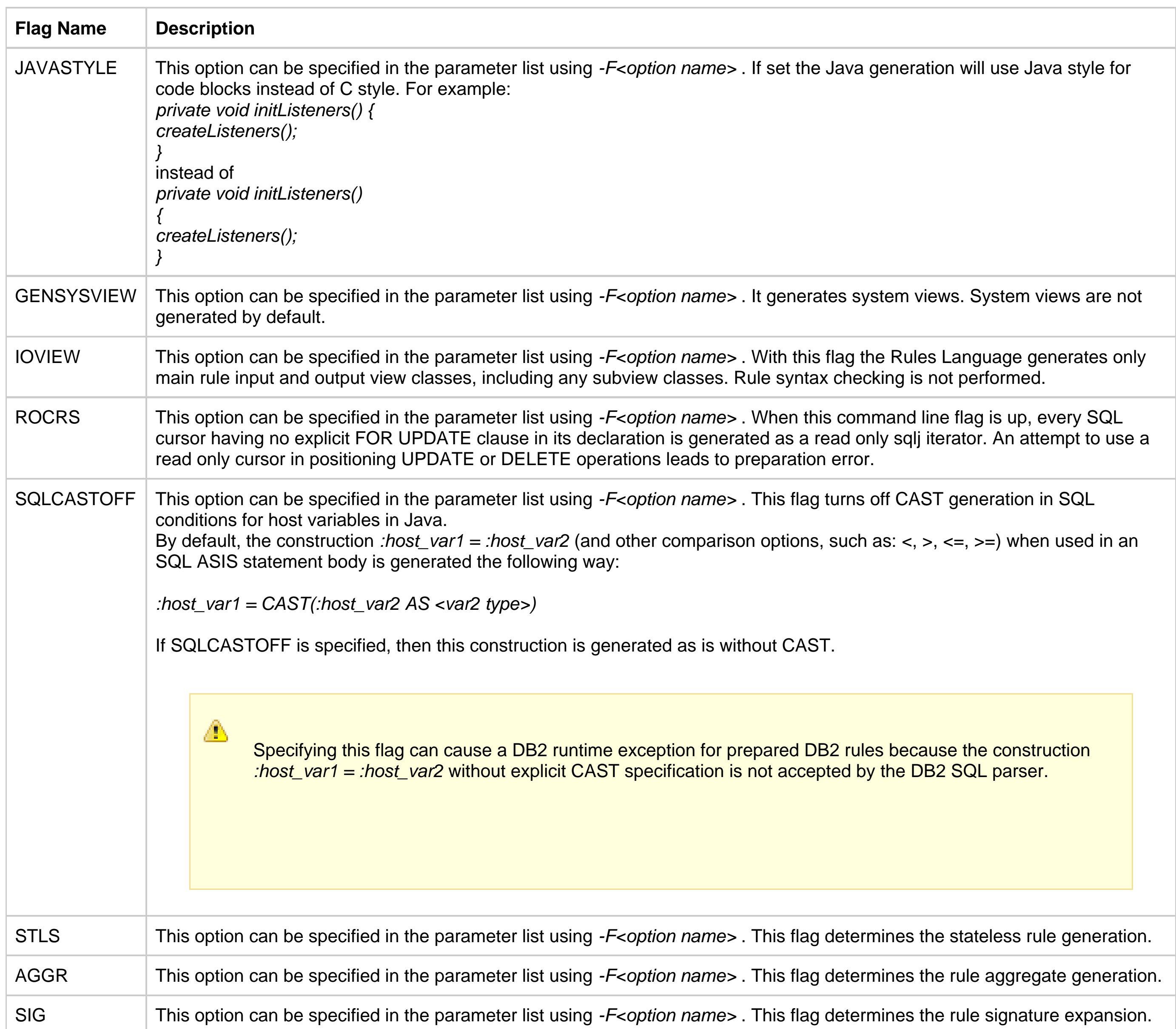
- -yd
- -F<flag_name> for Java generation
- -KNN

-yd

This parameter disables the generation of debugging information.

-F<flag_name> for Java generation

The following flag_name can be specified in the parameter list using *-F<flag_name>:*



Flag Name	Description
JAVASTYLE	This option can be specified in the parameter list using *-F<option name>* . If set the Java generation will use Java style for code blocks instead of C style. For example: *private void initListeners() {* *createListeners();* *}* instead of *private void initListeners()* *{* *createListeners();* *}*
GENSYSVIEW	This option can be specified in the parameter list using *-F<option name>* . It generates system views. System views are not generated by default.
IOVIEW	This option can be specified in the parameter list using *-F<option name>* . With this flag the Rules Language generates only main rule input and output view classes, including any subview classes. Rule syntax checking is not performed.
ROCRS	This option can be specified in the parameter list using *-F<option name>* . When this command line flag is up, every SQL cursor having no explicit FOR UPDATE clause in its declaration is generated as a read only sqlj iterator. An attempt to use a read only cursor in positioning UPDATE or DELETE operations leads to preparation error.
SQLCASTOFF	This option can be specified in the parameter list using *-F<option name>* . This flag turns off CAST generation in SQL conditions for host variables in Java. By default, the construction *:host_var1 = :host_var2* (and other comparison options, such as: <, >, <=, >=) when used in an SQL ASIS statement body is generated the following way: *:host_var1 = CAST(:host_var2 AS <var2 type>)* If SQLCASTOFF is specified, then this construction is generated as is without CAST. Specifying this flag can cause a DB2 runtime exception for prepared DB2 rules because the construction *:host_var1 = :host_var2* without explicit CAST specification is not accepted by the DB2 SQL parser.
STLS	This option can be specified in the parameter list using *-F<option name>* . This flag determines the stateless rule generation.
AGGR	This option can be specified in the parameter list using *-F<option name>* . This flag determines the rule aggregate generation.
SIG	This option can be specified in the parameter list using *-F<option name>* . This flag determines the rule signature expansion.

-KNN

In Java generation, this parameter sets the threshold value for decimal variable length which controls the data type and transformation routines used for host variable generation. The resulting code is DBMS- dependent. Current used types are listed in the following table:

DBMS vs DEC length	**<=NN**	**>NN**
DB2	java.math.BigDecimal	java.lang.String
Oracle	java.math.BigDecimal	java.math.BigDecimal
Default	double	double

Processing Order for Parameters

All settings are processed in the following order:

1. All default values are set.
2. The command line is parsed to determine the location of the INI file, the target language, and the environment if they are different from the default settings.
3. The INI file and platform independent sections are read. If the target platform was not specified on the command line and a value is specified in the INI file, it is used.
4. Language depended sections are processed.
5. All other command line parameters are processed. The parameters are read from the left to right and processed one by one. The parameter processed last takes precedence. In most cases, parameters are not verified against platform, target language, conflicts, or overwriting.

Code Generation Limitations

Rules Language generates programs in C, Java, or COBOL languages. There are certain limitations imposed to the generated programs by corresponding C, Java or COBOL compilers.
General restriction, which applies to all platforms and languages, concerns the ability to nest statements (IF, CASEOF, DO statements, expressions and conditions with brackets, including nested functions calls). While it is not possible to give the exact limit, 30 nested statements are always supported.

Java Restrictions

Java compiler imposes restriction on number of fields of view. It is not possible to give the exact limit, but it is not recommended to have more than 2000 fields in view.

C Restrictions

Microsoft *(R)* 32-bit C/C++ Optimizing Compiler has limitation on string literal - its length cannot exceed 2048 characters. SQL preprocessor converts each SQL ASIS block to a string literal; this imposes limitation on SQL ASIS block - there can't be more than 2048 symbols, including white-space.

Supported Codepages

The RC2_CODEPAGE parameter of the Hps.ini file defines the codepage in both ClassicCOBOL and OpenCOBOL on the host. Several string functions use the codepage setting when working with MIXED or DBCS values. For example, the function UPPER(DBCS) uses this setting. The following is a list of all the codepage names for Japanese and Korean locales:

- Ja_JP.IBM-290
- Ja_JP.IBM-930
- Ja_JP.IBM-939
- Ja_JP.IBM-1027
- Ja_JP.IBM-1390
- Ja_JP.IBM-1399
- Ko_KR.IBM-933
- Ko_KR.IBM-1364

For a list of all the possible values for codepages refer to the IBM *OS/390 C/C++ Programing Guide (Appendix D).*

Reserved Words

Reserved words have special meaning in the Rules Language. Do not use these words to name entities or variables in the application. Using a reserved word results in syntax errors when preparing the rule.

For applications written prior to HPS 5.4, you can disable some of the keywords by including the line PARAM=-yk in the [CodeGenParameters] section of the Hps.ini file, or with the PRAGMA KEYWORD statement. This allows you to compile the application with AppBuilder but prohibits you from using the functionality embodied in the new keywords.

The lists of reserved words differ for each target language. Refer to the following tables:

- Reserved Words for Java
- Reserved Words for C
- Reserved Words for ClassicCOBOL
- Reserved Words for OpenCOBOL

For keywords that can be disabled with PRAGMA KEYWORD, refer to the following tables:

- Keywords for Java that can be Disabled with PRAGMA KEYWORD
- Keywords for C that can be Disabled with PRAGMA KEYWORD
- Keywords for ClassicCOBOL that can be Disabled with PRAGMA KEYWORD
- Keywords for OpenCOBOL that can be Disabled with PRAGMA KEYWORD

For keywords that can be disabled with -YK, refer to the following tables:

- Keywords for Java that can be Disabled with -YK
- Keywords for C that can be Disabled with -YK
- Keywords for ClassicCOBOL that can be Disabled with -YK
- Keywords for OpenCOBOL that can be Disabled with -YK

Reserved Words for Java

AND	APPEND	ASIS	BASED
BREAK	BY	CASE	CASEOF
CATCH	CEIL	CHAR	CLASS
CLEAR	CLOSE	CLOSELOG	COMMIT
COMPONENT	CONTAINS	CONTINUE	CONVERSE
CURSOR	DATE	DAY	DAY_OF_WEEK
DAY_OF_YEAR	DBCS	DCL	DEC
DECLARE	DELETE	DETACH	DIV
DO	DOUBLE	ELSE	END
ENDCASE	ENDDCL	ENDDO	ENDIF
ENDPROC	ENDSQL	ENDTRY	EVENT
EXCEPTION	EXTERN	FALSE	FETCH
FLOAT	FLOOR	FOR	FRACTION
FROM	GETRULEIMPNAME	GETRULELONGNAME	GETRULESHORTNAME
GOTO	HANDLER	HIGH_VALUES	HOLD
HOURS	IF	IN	INDEX
INIT	INSERT	INSET	INSTANCE
INT	INTEGER	INTO	ISCLEAR
LIKE	LISTENER	LOC	LONGINT

LOW_VALUES	LOWER	MAP	MILSECS
MINUTES	MINUTES_OF_DAY	MIXED	MOD
MODULE	MONTH	NEST	NEW
NEW_TO_OLD_DATE	NEW_TO_OLD_TIME	NIL	NOT
NOWAIT	OBJECT	OCCURS	OF
OLD_TO_NEW_DATE	OLD_TO_NEW_TIME	OPEN	OPENLOG
OR	OTHER	OVERLAY	PERFORM
PIC	POINTER	PRAGMA	PRINT
PRINTER	PROC	PTR	REDEFINE
REDEFINES	REPLACE	REPORT	RESIZE
RETURN	RGB	ROLLBACK	ROUND
RTRIM	RULE	SECONDS	SECONDS_OF_DAY
SECTION	SET	SETDISPLAY	SETENCODING
SIZEOF	SMALLINT	SQL	START
STARTINTERVAL	STARTTIME	STRING	STRLEN
STRPOS	SUBHEADER	SUBSTR	SWITCH
TERMINAL	THROW	TIME	TIMESTAMP
TO	TRACE	TRANSACTION	TRUE
TRUNC	TRY	TYPE	UPPER
USE	VARCHAR	VERIFY	VIA
VIEW	VOID	WHILE	WINDOW
WITH	YEAR		

Keywords for Java that can be Disabled with PRAGMA KEYWORD

BREAK	CATCH	CLASS	CLOSE
CONTINUE	CURSOR	DECLARE	DOUBLE
ENDTRY	EXCEPTION	FALSE	FETCH
FLOAT	FOR	GOTO	HANDLER
HOLD	INTO	LISTENER	MODULE
NEW	OBJECT	OPEN	PTR
REDEFINE	REDEFINES	SET	STRING
SWITCH	THROW	TRUE	TRY
TYPE	VIA	VOID	WITH

Keywords for Java that can be Disabled with -YK

BREAK	CATCH	CLASS	CLOSE
CONTINUE	CURSOR	DECLARE	DOUBLE
ENDTRY	EXCEPTION	FALSE	FETCH
FLOAT	FOR	GOTO	HANDLER

HOLD	INTO	LISTENER	MODULE
NEW	OBJECT	OPEN	PTR
REDEFINE	REDEFINES	STRING	SWITCH
THROW	TRACE	TRUE	TRY
TYPE	VIA	VOID	WITH

Reserved Words for C

AND	ASIS	BASED	BREAK
BY	CASE	CASEOF	CATCH
CEIL	CHAR	CLASS	CLEAR
CLOSELOG	COMMIT	COMPONENT	CONTAINS
CONTINUE	CONVERSE	DATE	DAY
DAY_OF_WEEK	DAY_OF_YEAR	DBCS	DCL
DEC	DETACH	DIV	DO
DOUBLE	ELSE	END	ENDCASE
ENDDCL	ENDDO	ENDIF	ENDPROC
ENDSQL	ENDTRY	EVENT	EXCEPTION
EXTERN	FALSE	FLOAT	FLOOR
FOR	FRACTION	FROM	GOTO
HIGH_VALUES	HOURS	HPSColor	HPSError
HPSErrorMessage	HPSResetError	IF	IN
INDEX	INIT	INSET	INSTANCE
INT	INTEGER	ISCLEAR	LIKE
LOC	LOW_VALUES	LOWER	MAP
MILSECS	MINUTES	MINUTES_OF_DAY	MIXED
MOD	MODULE	MONTH	NEST
NEW_TO_OLD_DATE	NEW_TO_OLD_TIME	NOT	NOWAIT
OBJECT	OCCURS	OF	OLD_TO_NEW_DATE
OLD_TO_NEW_TIME	OPENLOG	OR	OTHER
OVERLAY	PERFORM	PIC	POINTER
PRAGMA	PRINT	PRINTER	PROC
PTR	REDEFINE	REDEFINES	REPORT
RETURN	RGB	ROLLBACK	ROUND
RTRIM	RULE	SECONDS	SECONDS_OF_DAY
SECTION	SET	SETDISPLAY	SETENCODING
SIZEOF	SMALLINT	SQL	START
STARTINTERVAL	STARTTIME	STRING	STRLEN
STRPOS	SUBHEADER	SUBSTR	SWITCH
TERMINAL	THROW	TIME	TIMESTAMP

TO	TRACE	TRANSACTION	TRUE
TRUNC	TRY	TYPE	UPPER
USE	VARCHAR	VERIFY	VIA
VIEW	VOID	WHILE	WINDOW
YEAR			

Keywords for C that can be Disabled with PRAGMA KEYWORD

BREAK	CATCH	CLASS	CONTINUE
DOUBLE	ENDTRY	EXCEPTION	FALSE
FLOAT	FOR	GOTO	MODULE
OBJECT	PTR	REDEFINE	REDEFINES
SET	STRING	SWITCH	THROW
TRUE	TRY	TYPE	VIA
VOID			

Keywords for C that can be Disabled with -YK

BREAK	CATCH	CLASS	CONTINUE
DOUBLE	ENDTRY	EXCEPTION	FALSE
FLOAT	FOR	GOTO	MODULE
OBJECT	PTR	REDEFINE	REDEFINES
STRING	SWITCH	THROW	TRACE
TRUE	TRY	TYPE	VIA
VOID			

Reserved Words for ClassicCOBOL

ADDR	AND	ASIS	BASED
BOOLEAN	BREAK	BY	CASE
CASEOF	CATCH	CEIL	CHAR
CLASS	CLEAR	CLOSELOG	COMMIT
COMPONENT	CONTAINS	CONTINUE	CONVERSE
CURSOR	DATE	DAY	DAY_OF_WEEK
DAY_OF_YEAR	DBCS	DCL	DEC
DECLARE	DETACH	DIV	DO
DOUBLE	ELSE	END	ENDCASE
ENDDCL	ENDDO	ENDIF	ENDPROC
ENDSQL	ENDTRY	EVENT	EXCEPTION
EXTERN	FALSE	FLOAT	FLOOR
FOR	FRACTION	FROM	GOTO
HIGH_VALUES	HOURS	IF	IN

INDEX	INIT	INSET	INSTANCE
INT	INTEGER	ISCLEAR	LIKE
LOC	LOW_VALUES	LOWER	MAP
MILSECS	MINUTES	MINUTES_OF_DAY	MIXED
MOD	MODULE	MONTH	NEST
NEW_TO_OLD_DATE	NEW_TO_OLD_TIME	NOT	NOWAIT
OBJECT	OCCURS	OF	OLD_TO_NEW_DATE
OLD_TO_NEW_TIME	OPENLOG	OR	OTHER
OVERLAY	PERFORM	PIC	POINTER
PRAGMA	PRINT	PRINTER	PROC
PTR	REDEFINE	REDEFINES	REPORT
RETURN	ROLLBACK	ROUND	RTRIM
RULE	SECONDS	SECONDS_OF_DAY	SECTION
SET	SETDISPLAY	SETENCODING	SIZEOF
SMALLINT	SQL	START	STARTINTERVAL
STARTTIME	STRING	STRLEN	STRPOS
SUBHEADER	SUBSTR	SWITCH	TERMINAL
THROW	TIME	TIMESTAMP	TO
TRACE	TRANSACTION	TRUE	TRUNC
TRY	TYPE	UPPER	USE
VARCHAR	VERIFY	VIA	VIEW
VOID	WHILE	WINDOW	YEAR

Keywords for ClassicCOBOL that can be Disabled with PRAGMA KEYWORD

BREAK	CATCH	CLASS	CONTINUE
CURSOR	DECLARE	DOUBLE	ENDTRY
EXCEPTION	FALSE	FLOAT	FOR
GOTO	MODULE	OBJECT	PTR
REDEFINE	REDEFINES	SET	STRING
SWITCH	THROW	TRUE	TRY
TYPE	VIA	VOID	

Keywords for ClassicCOBOL that can be Disabled with -YK

BREAK	CATCH	CLASS	CONTINUE
CURSOR	DECLARE	DOUBLE	ENDTRY
EXCEPTION	FALSE	FLOAT	FOR
GOTO	MODULE	OBJECT	PTR
REDEFINE	REDEFINES	STRING	SWITCH
THROW	TRACE	TRUE	TRY

TYPE	VIA	VOID	

Reserved Words for OpenCOBOL

ADDR	AND	ASIS	BASED
BREAK	BY	CASE	CASEOF
CATCH	CEIL	CHAR	CLASS
CLEAR	CLOSELOG	COMMIT	COMPONENT
CONTAINS	CONTINUE	CONVERSE	DATE
DAY	DAY_OF_WEEK	DAY_OF_YEAR	DBCS
DCL	DEC	DETACH	DIV
DO	DOUBLE	ELSE	END
ENDCASE	ENDDCL	ENDDO	ENDIF
ENDPROC	ENDSQL	ENDTRY	EVENT
EXCEPTION	EXTERN	FALSE	FLOAT
FLOOR	FOR	FRACTION	FROM
GOTO	HIGH_VALUES	HOURS	IF
IN	INDEX	INIT	INSET
INSTANCE	INT	INTEGER	ISCLEAR
LIKE	LOC	LOW_VALUES	LOWER
MAP	MILSECS	MINUTES	MINUTES_OF_DAY
MIXED	MOD	MODULE	MONTH
NEST	NEW_TO_OLD_DATE	NEW_TO_OLD_TIME	NOT
NOWAIT	OBJECT	OCCURS	OF
OLD_TO_NEW_DATE	OLD_TO_NEW_TIME	OPENLOG	OR
OTHER	OVERLAY	PERFORM	PIC
POINTER	PRAGMA	PRINT	PINTER
PROC	PTR	REDEFINE	REDEFINES
REPORT	ROLLBACK	ROUND	RTRIM
RULE	SECONDS	SECONDS_OF_DAY	SECTION
SET	SETDISPLAY	SETENCODING	SIZEOF
SMALLINT	SQL	START	STARTINTERVAL
STARTTIME	STRING	STRLEN	STRPOS
SUBHEADER	SUBSTR	SWITCH	TERMINAL
THROW	TIME	TIMESTAMP	TO
TRACE	TRANSACTION	TRUE	TRUNC
TRY	TYPE	UPPER	USE
VARCHAR	VERIFY	VIA	VIEW
VOID	WHILE	WINDOW	YEAR

Keywords for OpenCOBOL that can be Disabled with PRAGMA KEYWORD

BREAK	CATCH	CLASS	CONTINUE
DOUBLE	ENDTRY	EXCEPTION	FLOAT
FALSE	FOR	GOTO	MODULE
OBJECT	PTR	REDEFINE	REDEFINES
SET	STRING	SWITCH	THROW
TRUE	TRY	TYPE	VIA
VOID			

Keywords for OpenCOBOL that can be Disabled with -YK

BREAK	CATCH	CLASS	CONTINUE
DOUBLE	ENDTRY	EXCEPTION	FALSE
FLOAT	FOR	GOTO	MODULE
OBJECT	PTR	REDEFINE	REDEFINES
STRING	SWITCH	THROW	TRACE
TRUE	TRY	TYPE	VIA
VOID			

Decimal Arithmetic Support

For backwards compatibility, AppBuilder has three different arithmetic implementations:

- Calculator Arithmetic — decimal arithmetic introduced in Seer*HPS 5.4.0 and implemented in AppBuilder. This arithmetic is not supported on the mainframe platform.
- COBOL Arithmetic — decimal arithmetic introduced and implemented in Seer*HPS 5.4.1 that conforms to COBOL rules for calculating the precision of intermediate results.
- Compatible Arithmetic — arithmetic that uses two sets of arithmetic functions, one for constant expression evaluation and another for runtime calculations.

See Specific Considerations for C, Specific Considerations for ClassicCOBOL, and Restrictions on Features by Target Language for arithmetic support platform specifics.

The following topics are also discussed in this chapter:

- Native versus Runtime Support Calculations
- Platform Support
- Implementation of DIV and MOD
- Mapping to and from Numeric Data Items
- Overflow Returned

The evaluation of every formula is divided into the sequence of basic operations, such as binary and unary operations, and standard and user-defined function calls. Operator precedence and parentheses control the order of the operations. Each operation produces an intermediate result (IR).

Native versus Runtime Support Calculations

Calculation of arithmetic expressions during rule execution can be performed using target language (C or COBOL) arithmetic operations or using routines included in runtime support libraries. These two methods are referred to as *native* and *runtime* support calculations.

Native calculations are somewhat faster, but they cannot always be used. For example, C does not support operations with long DECIMAL values.

If all of the following conditions are true for an expression in the rule, then native calculations are used:

- All the operands in the expression and destination field (if present) are of native type (The set of native types is different for each

implementation of runtime arithmetic support.)
- The expression does not contain the mathematical functions CEIL, FLOOR, ROUND, or TRUNC.
- Local procedures or object methods returning non-native types are not used.
- There are no division (/), exponentiation (**), INSET, DIV, or MOD operations in the expression.

Refer to Error Handling for important issues about the differences in the error handling during native calculations.

Platform Support

Different types of arithmetic are supported on the PC Platform and Mainframe Platform.

PC Platform

On a PC platform, all arithmetic modes are fully supported and implemented. Arithmetic mode is switched by the DECIMAL_ARITHMETIC_MODE parameter in the AP <*platform*> section of the HPS.INI initialization file. This parameter can have the following values:

- CALCULATOR - for *calculator* arithmetic
- COBOL - for COBOL arithmetic
- COMPATIBILITY - for *compatible* arithmetic

Mainframe Platform

COBOL arithmetic mode is the only fully-supported arithmetic on the mainframe. However, you can choose to perform constant folding using any of the three described arithmetic modes: *calculator*, COBOL, or *compatible*.

Calculator Arithmetic

The primary difference between *calculator* and COBOL arithmetic is the rules for calculating intermediate result precision.

Calculator arithmetic provides intermediate result precision to 63 (sixty-three) decimal digits independent of the precision of the operands involved. If the result has more than 63 digits, it is truncated using the rules explained below.

Example: Calculator Arithmetic

Assume that the intermediate result of an arithmetic operation has I integer and D fractional places:

- If `I > 63`, then an overflow error occurs.
- If `I <= 63`, then if `I + D > 63`, then `I + D - 63` digits of the fractional part is truncated.

In the following Rules code:

```
DCL
   D3100, D3100A DEC(31);
   D3131, D3131A DEC(31,31);
ENDDCL
*> case 1 <*
MAP 10**30 TO D3100
MAP D3100*D3100*D3100 TO D3100A *> Case 1: Overflow <*
*> case 2 <*
MAP 10**(-30) TO D3131
MAP D3131*D3131*D3131 TO D3131A *> Case 2: D3131A = 0 <*
RETURN
```

In case 1, the result of the first multiplication is `10**60`, and the result of the second one is `10**90`, but `I = 90 > 63`, so an overflow occurs.

In case 2, the result of the first multiplication is `10**(-60)`, and the result of the second one is `10**(-90)`. Here `I = 0`, and `D = 90`, so `90 - 63 = 27` digits are truncated. As remaining digits are zeroes, the result is zero.

Native Types in Calculator Arithmetic

Native types in *calculator* arithmetic are INTEGER and SMALLINT on all platforms.

Calculator arithmetic does not use native COBOL arithmetic for DECIMAL data type. Runtime procedures are used instead, and performance is sacrificed for increased precision when using *calculator* rules on the mainframe.

Constant Expressions in Calculator Arithmetic

Constant expressions are computed during compile time. If an overflow occurs during these computations, an error message is generated. In any case, when the integer part is truncated, an error message is issued. If the fractional part is truncated, only a warning message is generated.

When the integer part of the result has more than 31 digits:

- A warning message is issued if the constant expression is a part of an arithmetic formula containing non-constant operands.
- An error message is issued if, for example, there are only numeric literals and the whole expression can be calculated during compile time.

COBOL Arithmetic

COBOL arithmetic support includes these topics:

- Native Types in COBOL Arithmetic
- dmax Parameter
- Truncation Rules for Intermediate Result
- Length and Scale of Function Results
- Constant Expressions in COBOL Arithmetic

Intermediate result precision in the COBOL arithmetic rules depends on the precision of the operands and the precision of the destination field.

Assume that the operands of the binary operation have i1 or i2 integer places and d1 or d2 decimal places, respectively. Then, the intermediate result has the number of the integer and decimal places (ir and dr, respectively) as shown in the following table.

COBOL Arithmetic Intermediate Results

Operation	Integer Places	Decimal Places
+ or -	(i1 or i2) + 1, whichever is greater	d1 or d2, whichever is greater
*	i1 + i2	d1 + d2
/	i1 + d2	Dmax
DIV	i1 + d2	0
MOD	i1+i2+d2+1	Max(d1,d2)
**	63 – dr	Max(d1,d2,dmax)
Math functions	See Length and Scale of Function Results.	

Operation MOD is implemented as a composition of 3 operations: *op1 - (op1 div op2) * op2*

Native Types in COBOL Arithmetic

Native types in COBOL arithmetic are listed in the following table:

Native types in COBOL arithmetic

Platform	Types
PC platform	INTEGER or SMALLINT
MainFrame platform	INTEGER, SMALLINT and DEC or PIC values with length less than extended precision threshold (*EPT*). If native C or COBOL arithmetic is used, it is *not* possible to detect whether an overflow has occurred.

dmax Parameter

The parameter, ***dmax*** is a precision parameter that is defined for the expression according to the rules outlined below. Note that ***dmax*** is calculated for an expression "as a whole". For example, all subexpressions of the MAP statement source have the same ***dmax***.

Calculation for ***dmax*** is done separately (independent of the enclosing expression) for:

- The source of the MAP statement
- Arguments to user procedure calls (even if used inside an expression)
- Arguments to standard *non-math* functions

Arguments of standard math functions (ROUND, TRUNC, CEIL and FLOOR) are processed as follows: the first argument is considered a subexpression of the enclosing expression and ***dmax*** for the second argument is calculated separately.

For an expression, ***dmax*** is calculated as the maximum of *scales of operands* (as defined below) and the *scale of the target of an expression* (the target of a MAP statement or procedure parameter).

The operand of an expression can be: data fields, constants, user procedure calls, and standard functions. The scale for these operands is:

Data fields	The scale of the data field. For integer variables, it is 0 (zero).
Constants	The scale of the constant (number of significant digits after decimal point).
User procedures	The scale of the return value from procedure declaration.
Standard functions	The result scale (see Length and Scale of Function Results).

Truncation Rules for Intermediate Result

The following remarks concerning the truncation rules apply only to ClassicCOBOL. Truncation rules for the intermediate result in COBOL arithmetic depend on whether the expression requires extended precision. To determine this, the extended precision threshold *(EPT)* is used. The extended precision threshold can be set using rule preparation parameter -K<*value*>, EPT default value is 15.

An Expression does not require extended precision if *all* the following conditions are met:

- All the operands of the expression and result field are SMALLINT, INTEGER, or decimals having no more than extended precision threshold decimal places.
- The expression does not contain the mathematical functions CEIL, FLOOR, ROUND, or TRUNC.
- Local procedures or object methods returning decimals longer than extended precision threshold are not used.
- There are no division (/), exponentiation (**), INSET, DIV and MOD operations in the expression.

The following table indicates when intermediate results might get truncated (if an expression requires extended precision then *N* is equal to 63, otherwise *N* is equal to 31):

Intermediate Result Truncation Rules

Value of i + d	Value of d	Value of i + dmax	Action taken
<=N	Any value	Any value	i integer and d decimal places are carried
>N	<=dmax	Any value	N-d integer and d decimal places are carried
	>dmax	<=N	i integer and N-i decimal places are carried
		>N	N-dmax integer and dmax decimal places are carried

Length and Scale of Function Results

Rules Language supports four mathematical functions that mathematically modify an expression:

- CEIL
- ROUND
- TRUNC
- FLOOR

These functions use one or two parameters:

- The value to be modified
- The significant number of digits to which the function applies

  This is a positive value referring to the digits to the right of the decimal point - zero referring to the digit to the immediate left of the decimal point and a negative value referring to digits farther to the left of the decimal point.

The data type of the returned value for any of these functions is DEC. Refer to Functions for detailed descriptions and examples of the mathematical functions used in Rules Language.

Consider these two important cases. In the first case, the second argument of the math function is a variable or an expression. In the second case, the second argument is a constant or constant expression. In the first case, the length and the scale of the result cannot be accurately predicted during preparation time. In the second case, they can be predicted with reasonable accuracy.

Here **i**, **d**, **i1** and **d1** denote integer and decimal places for the result and the first operand of the math function, respectively. In the first case, where the second operand of a math function is a variable:

i = N-d1

d = d1

(See Intermediate Result Truncation Rules for the definition of **N**.)

In the second case, where the second operand of a math function is a constant with value **C** (or a constant expression giving integer result C):

If C <= 0 then i = max(i1,|C|), d=0

If C > 0 then i=i1, d = min(d1,|C|)

The values of **i** and **d** are used in the calculations of **dmax** and for the calculations of length and scale of operands in compound expressions.

An error is reported at compile time if **C** < -31, and an overflow situation occurs if the second operand is less than -31.

Overflow Conditions

Since all values of i and d are calculated at compile time, the following can occur:

- If the actual result of computations has an integer part longer than intermediate results precision calculated according to rules described in COBOL Arithmetic Intermediate Results, an overflow error occurs.
- If the result of the operation is assigned to any field, then its value is truncated according the destination data type.
- If the destination field has fewer decimal positions for the integer part than for the intermediate result, an overflow error occurs.
- If the destination field has fewer decimal places for the fractional part than the intermediate result has, then the fractional part is truncated according to the rules in Intermediate Result Truncation Rules and no error code is set.

Example: Overflow Conditions

The following example clarifies the concepts of overflow conditions (in this example, EPT=15).

```
DCL
D1500, D1500A DEC(15);
D1510, D1510A DEC(15,10);
I INTEGER;
ENDDCL
MAP 10**14 TO D1500
MAP D1500*D1500*D1500 TO D1500A
*> Overflow (result of operation has too long integer part) <*
MAP D1500 TO D1510
*> Overflow (integer part of source is longer than that of a target) <*
MAP -50 TO I
MAP ROUND(D1500,I) TO D1500A
*> No overflow - 0 is a correct result <*
MAP CEIL(D1500,I) TO D1500A
*> Overflow (result of math function, 10**50 , is too long) <*
```

Constant Expressions in COBOL Arithmetic

Constant expressions are computed during compile time.

An error message is generated if:

- Division by zero occurs during these computations.
- The integer part is truncated.

However, if the fractional part is truncated, only a warning message is generated.

When the integer part of the result has more than 31 digits:

- A warning message is returned if the constant expression is a part of an arithmetic formula containing non-constant operands.
- An error message is returned if there are only numeric constants in the expression, and the whole expression is calculated during compile time.

Compatible Arithmetic

AppBuilder uses two sets of arithmetic functions: one for [Runtime Calculations](#) and one for [Constant Expression Evaluation in Compatible Arithmetic](#). See also [Division by Zero](#).

Runtime Calculations

The original arithmetic uses the value q_max to calculate precision of the intermediate results. q_max is calculated according to the following rules:

- If an expression is a source of a MAP statement, q_max is equal to the scale of destination.
- If an expression is a part of the condition and the other operand of the condition is a constant or a variable, then q_max is equal to its scale.
- Otherwise, q_max is equal to 0.

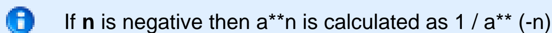
Intermediate result is calculated in two steps:

1. **Length** and **scale** of an intermediate result, **dmax** and **maxlen** are calculated.

   In the following table:
   - **I**, **S** are the length and the scale of the intermediate result,
   - **i1**, **s1** are the length and the scale of the first operand,
   - **i2**, **s2** are the length and the scale of the second operand,
   - **n** is the second operand of exponentiation.

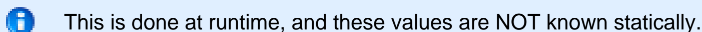
   Compatible Arithmetic Intermediate Results

Operations	I	S	Dmax	Maxlen
+/-	max(i1,i2)+1	max(s1,s2)	max(s1,s2,q_max)	64
*	i1+i2	s1+s2	max(s1,s2,q_max)	64
/	i1+s2	max(s1,q_max)	max(s1,q_max)	64
**	i1*|n|	s1*|n|	max(s1,q_max)	32 if n is even, 31 if n is odd

   
   If **n** is negative then a**n is calculated as 1 / a** (-n)

2. Intermediate result is truncated.

   If i+s > maxlen, truncation of the intermediate result is performed according to the following rules:
   - If s <= dmax then i is set to maxlen - s
   - Else if i + dmax < maxlen then s is set to maxlen - i
   - Else i is set to maxlen - dmax and s is set to dmax.

   
   This is done at runtime, and these values are NOT known statically.

Constant Expression Evaluation in Compatible Arithmetic

- [Constant Expressions Using Compatible Arithmetic - PC](#)
- [Constant Expression Using Compatible Arithmetic - Mainframe](#)

Constant Expressions Using Compatible Arithmetic - PC

Constant folding on the PC platform occurs only if the following conditions are satisfied:

- An expression is a source of a MAP statement, a parameter of a standard function, a FROM clause index, or a variable subscript. When an expression is not a source of MAP statement, its target is integer variable.

- All operands are constants in the whole expression (with the exception of 0**expr and expr**0 – these expressions are always treated as constants 0 and 1 respectively).
- An expression does not contain CEIL, FLOOR and TRUNC.
- Extended precision is not required for calculating an expression.

Division operator is handled by different rules. See Division by Zero for more details.

Use the following rules to determine whether or not an expression requires extended precision:

1. A constant requires extended precision if its length is greater than EPT (see COBOL Arithmetic and Truncation Rules for Intermediate Result). An operation requires extended precision if any of the operands require extended precision.
2. To determine whether an expression requires extended precision, its value is calculated according to the rules described in the next section. If the result is greater than 10**EPT, an expression require extended precision. If the length of the target of a MAP statement is longer than EPT digits, the expression require extended precision.

The constant folding is performed using native C language type **double** through C standard library functions. Because *q_max* parameter is not used, constant arithmetic computation results might differ from those of *compatible* calculations. The results of *compatible* computations might differ from those of *calculator* arithmetic in the 15th digit after the decimal point. The default value of extended precision threshold is 15.

Constant Expression Using Compatible Arithmetic - Mainframe

In *compatible* arithmetic, rules for constant folding on the mainframe are generally similar to those on the PC platform, with the following exceptions:

1. The operations ** and MOD are never folded.
2. Subscripts are never folded.
3. If the first argument of ROUND does not require extended precision, the second argument is treated as a separate expression and folded (if it can be folded) in the same way as a source of MAP into INTEGER, regardless of first argument being or not being constant.

Thus `1+2` in

```
MAP ROUND(D1510, 1+2) TO D1610
```

is computed at compile time (if ETP >= 15), but 1 MOD 2 in

```
MAP ROUND(D1510, 1 MOD 2) TO D1610
```

is not computed at compile time.

Division by Zero

If, during a rule preparation in *compatible* arithmetic, a division (/ or DIV) is encountered in any expression, then the attempt to calculate the divisor is made regardless of rules described above. The divisor is calculated if it does not contain any variables and there are no extended precision operands in expression.

Results upon division by zero that occur during rule execution in all arithmetic modes are described in:

- Division by Zero Using Compatible Arithmetic - PC
- Division by Zero Using Compatible Arithmetic - Mainframe

Division by Zero Using Compatible Arithmetic - PC

If division by zero occurs in *native calculations*, rule execution stops with a system error in all three arithmetic modes.

In runtime support calculations, behavior is triggered by D0_CHECK key of [AE Runtime] section of HPS.INI.

If D0_CHECK equals YES (all upper case; no blanks allowed), then rule execution stops with a system error.

Otherwise:

- in *calculator* and COBOL arithmetic, the result of the operation is overflow and HPSError is set
- in *compatible* arithmetic, the result of the operation is overflow and HPSError does *not* change

Division by Zero Using Compatible Arithmetic - Mainframe

Only COBOL arithmetic is implemented on the mainframe platform. In case of division by zero, rule execution stops with system error in both native and runtime support calculations.

Error Handling

If an error occurs during a constant expression calculation, an error message is issued as a result of the rule preparation.

On the mainframe, both Division-by-Zero Error - PC and Overflow Error produce system errors. The overflow value is either a DEC or PIC value filled with a symbol '*' (length is equal to the length of the variable), or a INTEGER or SMALLINT value equal to 0. Any operation with a DEC overflow value results in the overflow value; however, this is not true for an INTEGER and SMALLINT overflow value. Division-by-zero and overflow runtime errors on the PC workstation are described in Division-by-Zero Error Results and Overflow Error Results.

Division-by-Zero Error - PC

Division-by-Zero Error Results

Zero is a result of:	On a PC you receive:
Runtime arithmetic calculation	HpsError is set to the corresponding error code, and the rule continues executing. The result of computation is an overflow value.
Native expression	C runtime error

Use the D0_CHECK key of the [AE Runtime] section of the HPS.INI file to change the default result for the division-by-zero in a runtime arithmetic calculation exception.

If the D0_CHECK key is set to YES (all capital letters), the rule execution stops with the system exception Division-by-zero whenever the divisor in the MOD, DIV or / operation is equal to zero.

Overflow Error

Overflow Error Results

Overflow is a result of:	On a PC you receive:
Runtime arithmetic calculation	HpsError is set to the corresponding error code, and the rule continues executing. The result of computation is an overflow value.
Native expression	No error code is set. Rule continues executing. The result of operation is unpredictable.

Use the OVERFLOW_CHECK key in the [AE Runtime] section of the HPS.INI file to change the default result for the overflow in a runtime arithmetic calculation exception.

When this key is set to YES (all capital letters), then the rule execution is stopped with the system exception.

There is no way to stop rule execution and report an error if native expression was generated.

Implementation of DIV and MOD

AppBuilder supports DIV and MOD operations with non-integer operands.

In *compatible* and *calculator* arithmetic, DIV is implemented as follows:

To calculate `A div B, A / B` is calculated and all digits after decimal point are truncated.

In *compatible* and *calculator* arithmetic, MOD is implemented as follows:

```
A mod B = A - B * (A div B)
```

In COBOL arithmetic, DIV is implemented as a division with scale of a result equal to 0 and MOD as in the *compatible* runtime and *calculator* arithmetic. Results of computations are the same for all of these implementations.

Example: DIV and MOD implementation

The following is the examples of DIV and MOD operations:

```
MAP 11   DIV 2   TO X   *> X = 5    <*
MAP 11   DIV 0.2 TO X   *> X = 55   <*
MAP 1.1  DIV 0.2 TO X   *> X = 5    <*
MAP 0.11 DIV 0.2 TO X   *> X = 0    <*
MAP 11   MOD 2   TO X   *> X = 1    <*
MAP 11   MOD 0.2 TO X   *> X = 0    <*
MAP 1.1  MOD 0.2 TO X   *> X = 0.1  <*
MAP 0.11 MOD 0.2 TO X   *> X = 0.11 <*
```

Mapping to and from Numeric Data Items

Different numeric data items have a different range of values. Overflows can occur during mappings to numeric data items. If the MAP target is not a native type variable, an overflow value is assigned to that data item. However, if the MAP target is a native type variable, the value assigned cannot be predicted. Specifically, on the mainframe, execution is terminated with a system error. On a workstation, the value assigned to the INTEGER or SMALLINT variable cannot be predicted.

Situations when error and warning messages are issued at compile time in processing MAP statements to and from these numeric data items are described in the following sections:

- INTEGER and SMALLINT
- DEC and PIC
- Expressions

INTEGER and SMALLINT

If a constant is assigned to an INTEGER or SMALLINT variable, no checking for overflows is performed.

If a variable is assigned to an INTEGER or SMALLINT variable, a warning is issued if its integer part is greater than 10 or 5 digits, respectively.

DEC and PIC

If a constant with decimal part present is assigned to the DEC or PIC variable that does not fit, the error about overflow or a warning about truncation is issued accordingly.

If an integer constant is assigned to a DEC or PIC variable that does not fit, the error is issued only if the constant absolute value is greater than or equals 2**32.

If a variable is assigned to a DEC or PIC variable that does not fit, a warning is issued.

Expressions

If an expression is assigned to numeric data item, no checking for overflows is performed.

Overflow Returned

In *calculator* and COBOL arithmetic, math functions return overflow only if the decimal part of the result is longer than 63 digits.

In *compatible* runtime, arithmetic math functions return 0 for variables and overflow result for constants if an overflow situation occurs.

Example: Overflow Return Function

Consider the following example:

```
DCL
   I INTEGER;
   D1000 DEC(10);
   D1001 DEC(10,1);
   D0201 DEC(2,1);
ENDDCL

MAP 1 TO I
MAP I/3+I/3+I/3+I/3 TO D1000  *> case A <*
MAP I/3+I/3+I/3+I/3 TO D1001  *> case B <*
MAP 1/3+1/3+1/3+1/3 TO D1000  *> case C <*

MAP 1 TO D0201
MAP D0201/3+ D0201/3+ D0201/3+ D0201/3 TO D1000  *> case D <*
MAP D0201/3+ D0201/3+ D0201/3+ D0201/3 TO D1001  *> case E <*
RETURN
```

Results (values MAPped in destinations) are described in the following table.

Overflow Return Results

	Compatible	Calculator	COBOL
A	0 *(q_max = 0, thus no digits after decimal point are kept)*	1 *(all 63 digits are calculated)*	0
B	1.2 *(q_max = 1, one digit is kept)*	1.3 *(all 63 digits are still calculated, providing more accurate result)*	1.2
C	1 *(constant computations – four C doubles are added together)*	1 *(same arithmetic functions used for constant and runtime computations)*	0
D	1 *(q_max=0, but dmax=1 for division)*	1	1
E	1.2	1.3	1.2

Rules Language Quick Reference and Syntax

This chapter provides a quick reference to the following Rules Language statements and syntax diagrams. For detailed information regarding each of these statements, refer to the chapters where they are discussed.

- Data Types Syntax
- Data Items Syntax
- Arithmetic Operators Syntax
- Functions Syntax
- Declaration Syntax
- Common Procedure Syntax
- Event Procedure Syntax
- Control Statements Syntax
- Assignment Statement Syntax
- Condition Operators Syntax
- Condition Statements Syntax
- Transfer Statements Syntax
- Macro Statements Syntax

The table below is a list of statements and their brief descriptions:

Rules Language Statements by categories (Continued)

Statements	Description
DATA TYPES	
BOOLEAN	The BOOLEAN data type holds a value either TRUE or FALSE. Refer to the BOOLEAN Data Type for more information.
CHAR	Use the CHAR data type for a fixed-length character data item. Refer to the Character Data Types for more information.

DATE	Use the DATE data type for a date data item. The value in the data item is the number of days past the date of origin. January 1, 0000 is the date of origin and has a date number of 1. The DATE variable has a length of four-bytes except for OpenCOBOL. Refer to the Date and Time Data Types for more information.
DBCS	The DBCS data type can contain only fixed-length, double-byte character set data items. Refer to the Character Data Types for more information.
DEC	Use the DEC data type to specify a decimal data item. The first integer value after a DEC keyword is the total length of the data item; the second integer value is the scale, indicating the number of places to the right of the decimal point. Refer to the Numeric Data Types for more information.
IMAGE	Use the IMAGE data type for a data item that holds a reference to a binary large-object file (BLOB). Refer to the Large Object Data Types for more information.
INTEGER	The INTEGER data type holds a four-byte integer data item that contains values between -2,147,483,648 and 2,147,483,647 inclusive. Refer to the Numeric Data Types for more information.
MIXED	The MIXED data type can contain double-byte characters and single-byte characters, in any combination thereof. Refer to the Character Data Types for more information.
OBJECT ARRAY	Use the array object (OBJECT ARRAY form) to declare an array as a locally-declared data item. Refer to the Array Object for more information.
OBJECT OBJECT POINTER	The OBJECT data type and OBJECT POINTER data type represent a non-typed reference to an object. It is supported only for generation to Java. OBJECT data type is equivalent to the OBJECT POINTER data type. Refer to the Object Data Types for more information.
PIC	Declaring a data item as an integer picture (PIC) creates a storage picture that structures numeric data according to the PIC data format. For example, a PIC data item declared with the storage picture S999V99 can contain numeric data from -999.99 to 999.99. Refer to the Numeric Data Types for more information.
SMALLINT	The SMALLINT data type holds a two-byte integer data item that contains values between -32,768 and 32,767 inclusive. Refer to the Numeric Data Types for more information.
TEXT	Use the TEXT data type for a data item that holds a reference to a large-object, text file. Refer to the Large Object Data Types for more information.
TIME	Use the TIME data type for a time data item. The value in the data item is the number of milliseconds past midnight. The TIME data type has a length of four-bytes except for OpenCOBOL. Refer to the Date and Time Data Types for more information.
TIMESTAMP	Use the TIMESTAMP data type for a time data item where you need greater precision than milliseconds. The TIMESTAMP data type has a length of 12 bytes except for OpenCOBOL. It consists of three independent sub-fields: <DATE>:<TIME>:<FRACTION> Refer to the Date and Time Data Types for more information.
VARCHAR	Use the VARCHAR data type for a variable-length character data item. Refer to the Character Data Types for more information.
DATA ITEMS	
Alias	An alias is a name assigned to a data item of OBJECT data type to be used in the Rules Language in place of a system ID to refer to an object.
Character Value	A character value can be a symbol associated with a character value, a field of a character data type, or a character literal. A character literal is a string of up to 50 characters enclosed in single or double quotation marks.
Numeric Value	A numeric value can be a symbol associated with a numeric value, a field of a numeric data type, or a numeric literal. A numeric literal is either an integer or a decimal number.
Symbol	Symbol's value is a constant. You can store character and numeric literals in the repository as symbol entities and group them into Sets.
Variable Data Item	A rule can use any view or field in its data universe as a variable data item. You can either define a variable in your repository to be used globally or declare it within a rule to be used locally within that rule.
View	A View is an object in the Information Model that defines a data structure you use within your rules.
ARITHMETIC OPERATORS	
+ (Addition)	Adds two expressions together.
- (Subtraction)	Subtracts its second expression from its first.
* (Multiplication)	Multiplies two expressions.

/ (Division)	Divides its first expression by its second.
** (Exponentiation)	Raises its first expression to the power of its second.
DIV (Integer division)	Returns the number of times the second operand can fit into the first.
MOD (Modulus)	Provides the remainder from an integer division operation.
+:= (Increment)	Adds the right operand to the variable, which is its left operand.
-:= (Decrement)	Subtracts its right operand from the variable, which is its left operand.
FUNCTIONS	
++ (Concatenation)	The ++ function returns the concatenation of the two input strings.
APPEND	In Java, the APPEND function appends the source_view to the target_view. Views must be identical in structure.
CEIL	The CEIL function returns the next number greater than the first expression to the significant number of digits indicated by the second expression.
CHAR	The CHAR function converts a value in a DATE or TIME field to a value in a CHAR field.
CHAR	The CHAR function supports conversion from numbers to character strings.
CHAR	In the DBCS-enabled version of AppBuilder, the CHAR function treats the character value as a CHAR data item. Refer to Double-Byte Character Set Functions for more information.
CLEARNULL in Java	The CLEARNULL function is available for Java only. This function takes a field or a view as an argument and clears the NULL flag of the field or every field in a view if it is applied to a view, without changing the value of the field.
DATE	The DATE function returns the date (or current system date if no argument) in a DATE field,
DAY	The DAY function returns the day of the month for that date in a SMALLINT field.
DAY_OF_WEEK	The DAY_OF_WEEK function returns the day of the week for that date in a SMALLINT field.
DAY_OF_YEAR	The DAY_OF_YEAR function returns the Julian day of the year for that date in a SMALLINT field.
DBCS	In the DBCS-enabled version of AppBuilder, DBCS function treats the character value as a DBCS data item. Refer to Double-Byte Character Set Functions for more information.
DECIMAL	The DECIMAL function converts character strings to numeric (decimal) values. Refer to the Numeric Conversion Functions for more information
DECR	The DECR function subtracts its second parameter from its first parameter and returns this modified first parameter.
DELETE	In Java, the DELETE function deletes occurrences of a view starting from the position given in the second argument.
FLOOR	The FLOOR function returns the next number less than the first expression to the significant number of digits indicated by the second expression.
FRACTION	The FRACTION function returns the number of picoseconds for that timestamp in an INTEGER field. On the host, this function returns the number of picoseconds. On workstations, this function always returns 0 because it is not feasible to obtain a unit of time smaller than a millisecond.
GET_ROLLBACK_ONLY in Java	The GET_ROLLBACK_ONLY function is available for Java only. This function returns a BOOLEAN value, indicating whether or not the only possible outcome of the transaction associated with the current thread is to roll back the transaction (TRUE) or not (FALSE).
HIGH_VALUES	The HIGH_VALUES function represents one or more characters that have the highest ordinal position in the collating sequence used.
HOURS	The HOURS function returns the number of hours since midnight for that time in a SMALLINT field.
HPSError	The HPSError function analyzes errors during program execution. Returns 0 if no error, or an integer value that represents the error code of the first error that occurs. Refer to the HPSError Function for more information.
HPSErrorMessage	The HPSErrorMessage function takes an error code as an argument and returns the text string containing a short description of the error condition. Refer to the HPSErrorMessage Function for more information.

HPSResetError	The HPSResetError function resets the error code to 0 after one or more error conditions have occurred. Refer to the HPSResetError Function for more information.
INCR	The INCR function adds its second parameter to its first parameter and returns the modified first parameter.
INSERT	In Java, the INSERT function inserts all occurrences of the source view (or the view itself if it is the plain view) at the specified position in the target view. Views must be identical in structure.
INT	The INI function converts character strings to numeric (integer) values. Refer to the Numeric Conversion Functions for more information
INT	The INT function converts the time or date in the specified TIME or DATE field, and returns a value in an INTEGER field.
ISNULL in Java	The ISNULL function is available for Java only. This function takes a field as an argument and returns a BOOLEAN value indicating whether the field is NULL or not.
LOC	The LOC function takes a view as an argument and returns its location in a CHAR (8) field.
LOW_VALUES	The LOW_VALUES function represents one or more characters that have the lowest ordinal position in the collating sequence used.
MILSECS	The MILSECS function returns the number of milliseconds past the second for that time in a SMALLINT field.
MINUTES	The MINUTES function returns the number of minutes past the hour for that time in a SMALLINT field.
MINUTES_OF_DAY	The MINUTES_OF_DAY function returns the number of minutes since midnight for that time in a SMALLINT field.
MIXED	In the DBCS-enabled version of AppBuilder, the MIXED function treats the character value as a MIXED data item. Refer to Double-Byte Character Set Functions for more information.
MONTH	The MONTH function returns the month of the year for that date in a SMALLINT field.
NEW	The NEW clause is used in the Java application development to create new instances of objects. Refer to the Creating a New Object Instance in Java for more information.
NEW_TO_OLD_DATE	The NEW_TO_OLD_DATE function converts the date in the specified DATE field, and returns a value in an INTEGER field.
NEW_TO_OLD_TIME	The NEW_TO_OLD_TIME function converts the time in the specified TIME field, and returns a value in an INTEGER field.
OCCURS	In Java, the OCCURS function returns the number of occurrences of a given *view* . For non-occurring views, it returns 0.
OLD_TO_NEW_DATE	The OLD_TO_NEW_DATE function converts the value in the specified INTEGER field, and returns a value in a DATE field.
OLD_TO_NEW_TIME	The OLD_TO_NEW_TIME function converts the value in the specified INTEGER field, and returns a value in a TIME field.
REPLACE	In Java, the REPLACE function replaces occurrences of the *target_view* with occurrences from the *source_view* , starting from the specified position.
RESIZE	In Java, the RESIZE function shrinks or expands the given occurring view to a new size.
ROUND	The ROUND function returns the number closest to the first expression to the significant number of digits indicated by the second expression.
RTRIM	The RTRIM function returns the input string with any trailing blanks removed.
SECONDS	The SECONDS function returns the number of seconds past the minute for that time in a SMALLINT field.
SECONDS_OF_DAY	The SECONDS_OF_DAY function returns the number of seconds since midnight for that time in an INTEGER field.
SET_ROLLBACK_ONLY in Java	The SET_ROLLBACK_ONLY function is available for Java only. This function modifies the transaction associated with the current thread so that the only possible outcome of the transaction is to roll back the transaction.
SETDISPLAY	The SETDISPLAY function supports the use of sets. The first argument is the name of a Lookup Table set, the second argument is the value to look up in the set, and the last argument (needed only for an Multiple Language Support application) is the language entity to use for getting the representation of the encoding (the second argument). This function returns a value in a CHAR (80) field.

SETENCODING	The SETENCODING function supports the use of sets. The first argument is the name of a Lookup Table set, the second argument is the representation to look up in the set, and the last argument (needed only for an Multiple Language Support application) is the language entity used for getting the display (the second argument). This function returns a value of the same type and length as the set in the first argument.
SIZEOF	The SIZEOF function takes a view as an argument and returns its data length in bytes in a field of type INTEGER.
STRLEN	The STRLEN function returns a positive integer that specifies the length of the input string, not counting any trailing blanks.
STRPOS	The STRPOS function searches for a second string in the first string and returns the position (zero or a positive value) from which the second string starts.
SUBSTR	The SUBSTR function returns a substring of the input string that begins at the position the first expression indicates for the length the second expression indicates.
TIME	The TIME function returns the time (or current system time if no argument) in a TIME field.
TIMESTAMP	The TIMESTAMP function returns a timestamp created from the current date and time in a TIMESTAMP field if no argument is supplied, or the value in a TIMESTAMP field, if arguments are supplied.
TRACE	The TRACE function can be used to output the Rules Language data items to an application trace file.
TRUNC	The TRUNC function returns a number that is the first expression with any digits to the right of the indicated significant digit set to zero.
UPPER and LOWER	The UPPER and LOWER functions return the input string with all alphabetic characters converted to uppercase or lowercase.
VERIFY	The VERIFY function looks for the first occurrence of a character in the first string that does not appear in the second string, and returns the position (zero or a positive integer) in characters.
YEAR	The YEAR function returns the year for that date in a SMALLINT or INTEGER field.
DECLARATIONS	
DCL ... ENDDCL	The DCL statement declares a variable data item or view locally. Refer to the Declarations for more information.
PROCEDURES	
HANDLER	In Java, the HANDLER clause enables event handlers for the specified object. Refer to the Event Handler Statement in Java for more information.
PROC ... ENDPROC	The PROC statement defines a procedure. Refer to the Event Handling Procedure for more information
CONTROL STATEMENTS	
> ... <	One or multiple line comment statement. Refer to the Comment Statement for more information.
//	One line comment statement. Refer to the Comment Statement for more information.
<object_name>.<method_name>	Invokes a method for an object. Refer to the ObjectSpeak Statement for more information.
COMMIT TRANSACTION	The COMMIT TRANSACTION statement commits changes to local and remote databases since the previous START TRANSACTION statement. Refer to the COMMIT TRANSACTION for more information.
POST EVENT	The POST EVENT statement posts a message to another application, or to a different rule in the same application. Refer to the Post Event Statement for more information.
PRAGMA ALIAS PROPERTY	The PRAGMA ALIAS PROPERTY statement defines an alias for a property. Refer to the PRAGMA ALIAS PROPERTY in Java for more information.
PRAGMA AUTOHANDLERS	In Java, PRAGMA AUTOHANDLERS determines whether or not event handlers for window objects are assigned automatically. Refer to the PRAGMA AUTOHANDLERS in Java for more information.
PRAGMA CENTURY	In OpenCOBOL, PRAGMA CENTURY allows you to specify the default century used in the conversion functions DATE(char) and CHAR(date). Refer to the PRAGMA CENTURY for OpenCOBOL for more information.
PRAGMA CLASSIMPORT	In Java, PRAGMA CLASSIMPORT makes the static fields and methods of Java classes available for the rule. Refer to the PRAGMA CLASSIMPORT in Java for more information.
PRAGMA COMMONHANDLER	In Java, PRAGMA COMMONHANDLER specifies the handler on any object's event using the same system ID (HPSID) within the rule scope. Refer to the PRAGMA COMMONHANDLER in Java for more information.

PRAGMA KEYWORD	The PRAGMA KEYWORD statement switches selected Rules Language keywords on or off. Refer to the PRAGMA KEYWORD for more information.
PRAGMA SQLCURSOR	In Java, PRAGMA SQLCURSOR specifies cursor field types without FETCHing all the cursor fields into the host variables. Refer to the PRAGMA SQLCURSOR in Java for more information.
ROLLBACK TRANSACTION	The ROLLBACK TRANSACTION statement rolls back changes to local and remote databases since the previous START TRANSACTION statement. Refer to the ROLLBACK TRANSACTION for more information.
SQL ASIS ... ENDSQL	The SQL ASIS statement embeds a SQL statement in a code to access a database. Refer to the SQL ASIS Support for more information.
START TRANSACTION	The START TRANSACTION statement starts a database transaction explicitly. Refer to the START TRANSACTION for more information.
ASSIGNMENT STATEMENTS	
CLEAR	The CLEAR statement sets the value of the specified variable to its initial value. Refer to the CLEAR Statement for more information.
MAP SET	The MAP or SET statement assigns a value to a variable data item. Refer to the Assignment Statements for more information.
OVERLAY	An OVERLAY statement copies the value of the first item into the second item. Refer to the OVERLAY Statement for more information.
CONDITION OPERATORS	
=	Is equal to. Refer to the Condition Operators for more information.
<>	Is not equal to. Refer to the Condition Operators for more information.
<	Is less than. Refer to the Condition Operators for more information.
<=	Is less than or equal to. Refer to the Condition Operators for more information.
>	Is greater than. Refer to the Condition Operators for more information.
>=	Is greater than or equal to. Refer to the Condition Operators for more information.
INSET	Is included in the set. Refer to INSET Operator for more information.
ISCLEAR	Is operand value equal to its initial value. Refer to ISCLEAR Operator for more information.
CONDITION STATEMENTS	
CASEOF ... CASE ... [CASE OTHER ...] ENDCASE	The CASEOF statement routes control among any number of groups of statements, depending on the value of a field. If none of the CASE clauses equal the value in the field and there is no CASE OTHER clause, no statements are executed. Refer to the CASEOF Statement for more information.
DO ... [FROM ...] [TO ...] [BY ...] [INDEX ...] [WHILE ...] ENDDO	The DO statement provides control for repetitive loops. If a DO statement contains a WHILE clause, any statements between DO and ENDDO are executed repetitively as long as the condition in the WHILE clause is true and the TO bound is not reached. When one of the conditions mentioned becomes false, control passes from the WHILE clause to the statement following the ENDDO. Refer to the DO Statement for more information.
IF ... [ELSE ...] ENDIF	The IF statement routes control between two groups of statements, depending on the truth or falsity of a condition. If the condition is false and there is no ELSE clause, no statements are executed. Refer to the IF Statement for more information.
TRANSFER STATEMENTS	
CONVERSE	The CONVERSE statement (without WINDOW or REPORT) has the effect of blocking a rule until an event is received. Refer to the CONVERSE for Global Eventing for more information.
CONVERSE REPORT	A rule can control the printing of a report by conversing a report entity. Refer to the CONVERSE REPORT Statement for more information.
CONVERSE WINDOW	The CONVERSE WINDOW statement causes the named window entity's panel to display on the screen, allowing a user to manipulate the window's interface and field data. In the rules code, execution remains on the CONVERSE WINDOW statement until an event is returned to the rule. Refer to the CONVERSE WINDOW Statement for more information.
PERFORM	The PERFORM statement allows you to invoke a procedure multiple times within a rule, rather than duplicating the statements of the procedure at multiple places in the rule. Refer to the PERFORM Statement for more information.

PROC RETURN	A PROC RETURN statement causes a procedure to return control to the point immediately after the point from which the procedure was invoked. Refer to the PROC RETURN Statement for more information.
RETURN	The RETURN statement sends processing control back to the calling rule before all lines in the called rule have been executed. Refer to the RETURN Statement for more information.
USE COMPONENT	The USE COMPONENT statement passes processing control to a component. Refer to the USE COMPONENT Statement for more information.
USE RULE	The USE RULE statement invokes another rule without any special instructions. If the called rule converses a window, all other windows in its application are removed before its window appears. Refer to the USE RULE Statement for more information.
USE RULE ... DETACH	The USE RULE ... DETACH statement invokes another rule and instructs the called rule to share control with the calling rule. Any window the called rule converses is still nested, but is modeless. Refer to the USE RULE ... DETACH Statement for more information.
USE RULE ... INIT	The USE RULE ... INIT statement initiates the execution of the called rule (and any rules and components it calls) and causes the called rule to run independently from the calling rule. Refer to the USE RULE ... INIT Statement for more information.
USE RULE ... NEST	The USE RULE ... NEST statement invokes another rule, and if the called rule converses a window, it instructs the called rule to overlay its window over other windows of the application that are currently visible. Refer to the USE RULE ... NEST Statement for more information.
MACROS	
CG_CGEXIT	The CG_CGEXIT statement breaks the process of translation with the return code *Return code.* Refer to the Exiting from Translation for more information.
CG_CHANGEQUOTE	Use the CG_CHANGEQUOTE statement to change the quote characters for macros. The default quote characters are <: to start, and :> to finish the quote. Refer to the Changing the Quote Characters in Macros for more information.
CG_DECR	The CG_DECR macro statement decrement an integer and return the result. Refer to the CG_INCR and CG_DECR for more information.
CG_DEFINE	Use the CG_DEFINE statement to define a macro. Refer to the Defining Macros for more information.
CG_ELSEIF CG_ELSEIFNOT	You can insert multiple CG_ELSEIF and CG_ELSEIFNOT statements to evaluate more conditions in one CG_IF statement. After the CG_IF statement evaluates to false, the CG_ELSEIF(*macro_name, value*) statement compares th *e macro_name* with the *value* , and if they are equal, all *statements* after CG_ELSEIF are processed. In the CG_ELSEIFNOT(*macro_name, value*) statement, th *e macro_name* is compared with the *value* , and if they are not equal, all *statements* after CG_ELSEIFNOT are processed. Refer to the CG_ELSEIF and CG_ELSEIFNOT for more information.
CG_EVAL	The CG_EVAL macro statement is used to perform more complex mathematical operations. It takes any *expression* and replaces it with the result. Refer to the CG_EVAL for more information.
CG_IF	With the CG_IF statement, th *e macro name* is compared with the *value* . If the *macro_name* and the *value* are equal, all *statements* after CG_ELSE are excluded from translation; if the *macro_name* and the *value* are not equal, only *statements* after CG_ELSE are processed. Refer to the Using Conditional Translation for more information.
CG_IFDEF	Use the CG_IFDEF statement to evaluate if a macro is defined or not. Refer to the Evaluating if a Macro Exists for more information.
CG_IFDEFINED	In the CG_IFDEFINED(*macro_name*) statement, the preprocessor analyzes to determine if *macro_name* has been defined. If it has been defined, all *statements* after CG_ELSE are excluded from translation; if it has not been defined, only *statements* after CG_ELSE are processed. Refer to the CG_IFDEFINED and CG_IFNOTDEFINED for more information.
CG_IFELSE	Use the CG_IFELSE statement to compare values and performs substitution based on the result of the comparison. Refer to the Comparing Values for more information.
CG_IFNOT	With the CG_IFNOT statement, th *e macro name* is compared with the *value* . If the *macro_name* and the *value* are NOT equal, all *statements* after CG_ELSE are excluded from translation. Refer to the Using Conditional Translation for more information.
CG_IFNOTDEFINED	In the CG_IFNOTDEFINED(*macro_name*) statement, if the *macro_name* has not been defined, all *statements* after CG_ELSE are excluded from translation; if it has been defined, only *statements* after CG_ELSE are processed. Refer to the CG_IFDEFINED and CG_IFNOTDEFINED for more information.
CG_INCLUDE	The CG_INCLUDE statement causes the compiler to process the file specified in the *file_name* parameter. Refer to the Including Files for more information.

CG_INCR	The CG_INCR macro statement increment an integer and return the result. Refer to the CG_INCR and CG_DECR for more information.
CG_INDEX	CG_INDEX returns the position of the *substring* in the *string* .
CG_LEN	CG_LEN returns the length of a string.
CG_SHIFT	This macro uses recursion to process parameters one by one, simulating the effect of looping. CG_SHIFT takes any number of parameters and returns the same list (each parameter quoted) after removing the first parameter. Refer to the Using Recursion to Implement Loops for more information.
CG_SUBSTR	CG_SUBSTR extracts some part of a *string* starting at the *from* position.
CG_UNDEFINE	Use the CG_UNDEFINE statement to undefine a macro. Refer to the Undefining a Macro for more information.
CG_CASEOF	The CG_CASEOF macro statement switches translation between any number of groups of statements, depending on the result of comparing *macro_name* with the *values* for each group. Refer to CG_CASEOF Statement for more information.
CG_IF with Boolean Condition	The CG_IF with Boolean Condition macro switches the translation depending of the truth or falsity of a condition. Refer to CG_IF Statement with Boolean Condition for more information.

Data Types Syntax

A Rules Language data item must be defined as a specific data type. Refer to the Data Types for more information about data types. The following data type syntax drawings are available under this section:

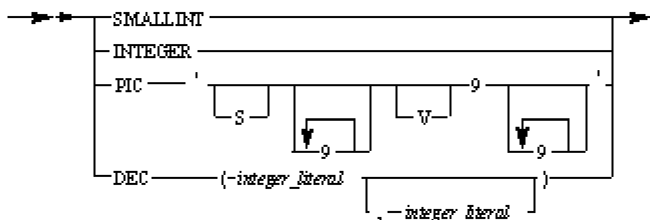
- BOOLEAN Data Type Syntax
- Numeric Data Types Syntax
- Date and Time Data Types Syntax
- Large Object Data Types Syntax
- Object Data Type Syntax
- Object Array Syntax
- Character Data Types Syntax

BOOLEAN Data Type Syntax

Refer to the BOOLEAN Data Type for more information.

Numeric Data Types Syntax

Refer to the Numeric Data Types for more information.



where:

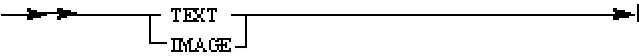
- *integer_literal* is an integer value specifying the total length of the data item and the scale.

Date and Time Data Types Syntax

Refer to the Date and Time Data Types for more information.

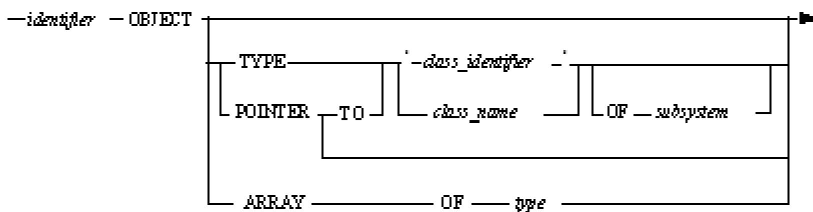
Large Object Data Types Syntax

Refer to the Large Object Data Types for more information.



Object Data Type Syntax

Refer to the Object Data Types for more information.



where:

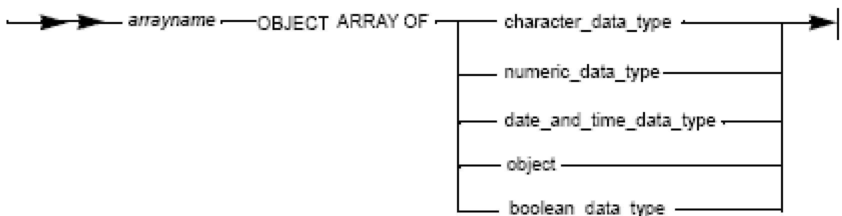
- *class_identifier* is a string that identifies the implementation of the class. Therefore, it might be the full Java class name for Java classes. The identification string is case-sensitive.
- *class_name* is a class name to be used in a rule. *Not* case-sensitive.
- *subsystem* is the group to which this object belongs.

The following subsystems are supported:

- GUI_KERNEL: the set of AppBuilder-supplied window controls.
- JAVABEANS: used for any Java class.
- *type* can be numeric, character, date and time, boolean, or object with certain limitations---see Array Object for more details.

Object Array Syntax

Refer to the Array Object for more information.

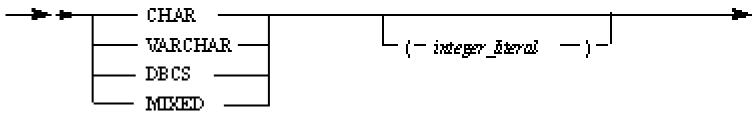


where:

- character_data_type---see Character Data Types.
- date_and_time_data_type---see Date and Time Data Types.
- numeric_data_type---see Numeric Data Types.
- object---see OBJECT. You can only have an array of non-typed objects, that is OBJECT ARRAY OF OBJECT.
- boolean_data_type---see BOOLEAN Data Type.

Character Data Types Syntax

Refer to the Character Data Types for more information.



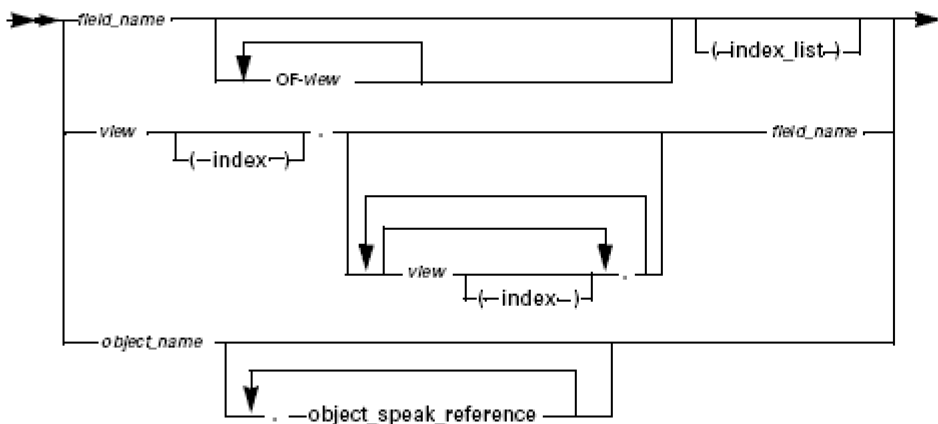
Data Items Syntax

A data item (or data element) is an individual unit of data that is processed by a rule. Refer to the Data Items for more information about data items. The following data items syntax drawings are available under this section:

- Variable Data Item Syntax
- View Data Item Syntax
- Character Value Syntax
- Numeric Value Syntax
- Symbol Syntax
- Alias Syntax

Variable Data Item Syntax

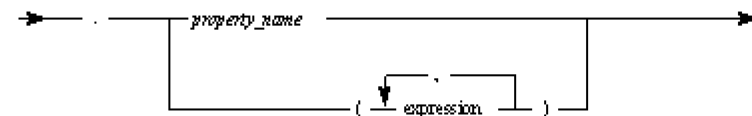
Refer to the Variable Data Item for more information.



where index_list is:

where index is:

where object_speak_reference is:



where *object_name* can be one of the following:

- The system identifier (HPSID) of the object
- The alias of the object---see Alias.
- An object---see Object Data Types.

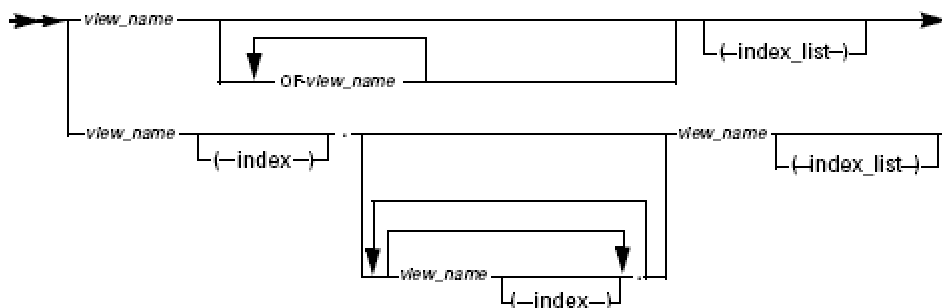
- An array---see Array Object.

where:

- expression---see Expression Syntax.
- numeric_expression---see Numeric Expressions.
- *view* ---see View.

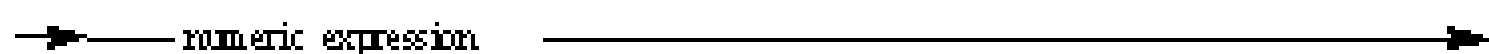
View Data Item Syntax

Refer to the View for more information.



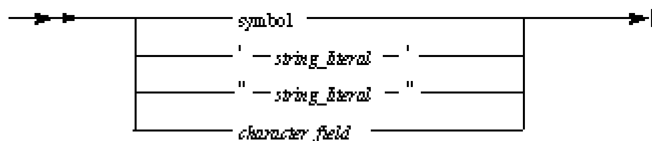
where index_list is:

where index is:



Character Value Syntax

Refer to the Character Value for more information.

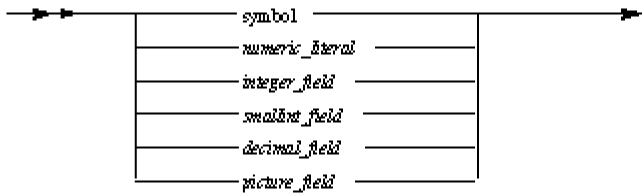


where:

- *character_field* is a variable data item of any character type.
- symbol---see Symbol.

Numeric Value Syntax

Refer to the Numeric Value for more information.

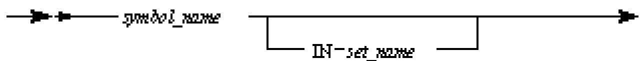


where:

- symbol---see Symbol.
- *integer_field* is a variable data item of INTEGER data type.
- *smallint_field* is a variable data item of SMALLINT data type.
- *decimal_field* is a variable data item of DEC data type.
- *picture_field* is a variable data item of PIC data type.

Symbol Syntax

Refer to the Symbol for more information.



Alias Syntax

Refer to the Alias for more information.

where

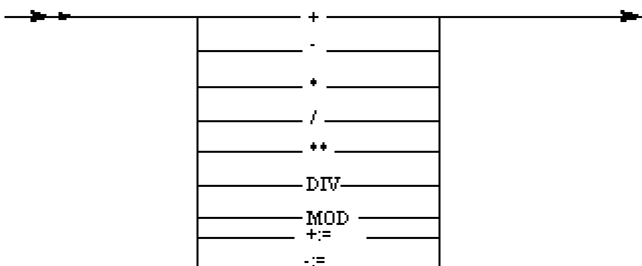
- *alias* is any valid Rules Language identifier.

system_identifier is the system identifier of an object declared in the panel file.

Arithmetic Operators Syntax

The Rules Language supports the basic arithmetic operations. Refer to the Arithmetic Operators for more information about arithmetic operators.

Arithmetic Operators Syntax



Functions Syntax

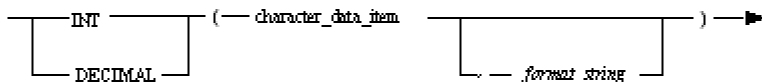
A function accepts one or more arguments, performs an action on them, and returns a value based on the action. Refer to the Functions for more information. The following functions syntax drawings are available under this section:

- Numeric Conversion Functions Syntax
- Mathematical Functions Syntax
- Date and Time Functions Syntax

- Character String Functions Syntax
- Support Functions Syntax
- Syntax for Creating a New Object Instance in Java
- Syntax for Dynamically-Set View Functions in Java

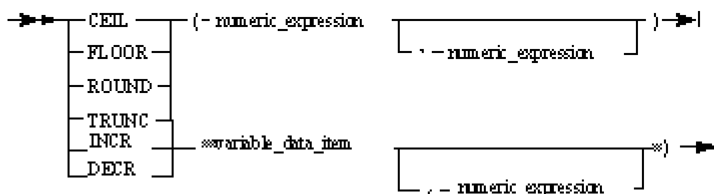
Numeric Conversion Functions Syntax

Refer to the Numeric Conversion Functions for more information.



Mathematical Functions Syntax

Refer to the Mathematical Functions for more information.

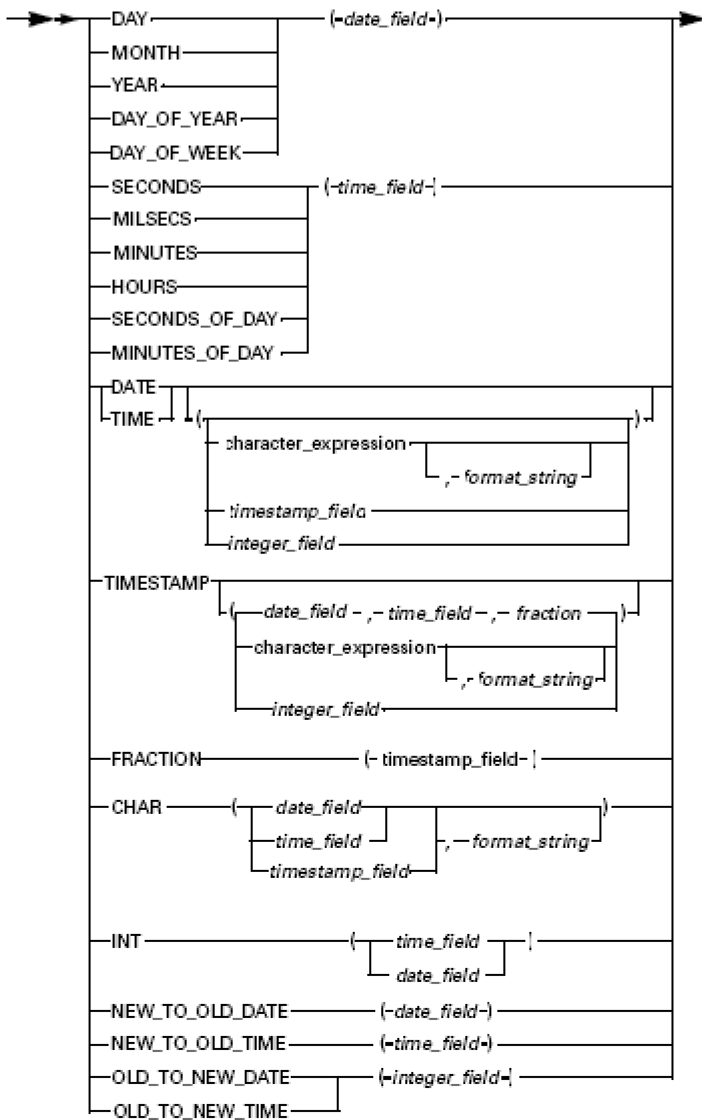


where:

- numeric_expression---see Numeric Expressions.
- variable_data_item is a variable data item of any numeric type.

Date and Time Functions Syntax

Use a DATE, TIME and TIMESTAMP function to obtain the current date, time, and timestamp, to format your data, or to convert a field from a date, time or timestamp data type to another data type. Refer to the Date, Time and Timestamp Functions for more information.

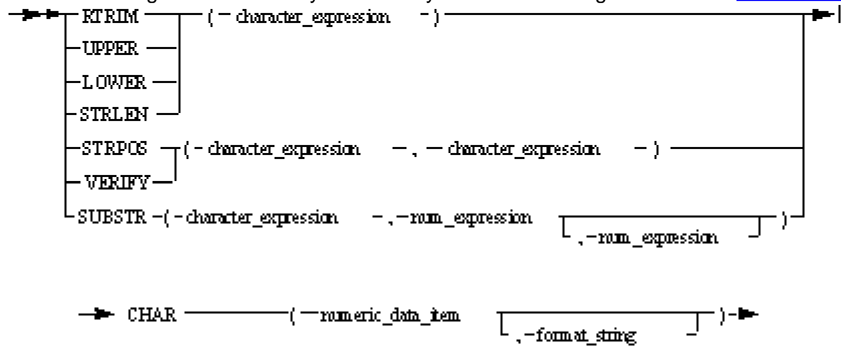


where:

- character_expression---see Character Expressions.
- *date_field* ---see Date and Time Data Types.
- *time_field* ---see Date and Time Data Types.
- *integer_field* ---see Numeric Data Types.
- *format_string* ---see Format String.
- timestamp_field---see Date and Time Data Types.

Character String Functions Syntax

Character string functions allow you to modify a character string. Refer to the Character String Functions for more information.

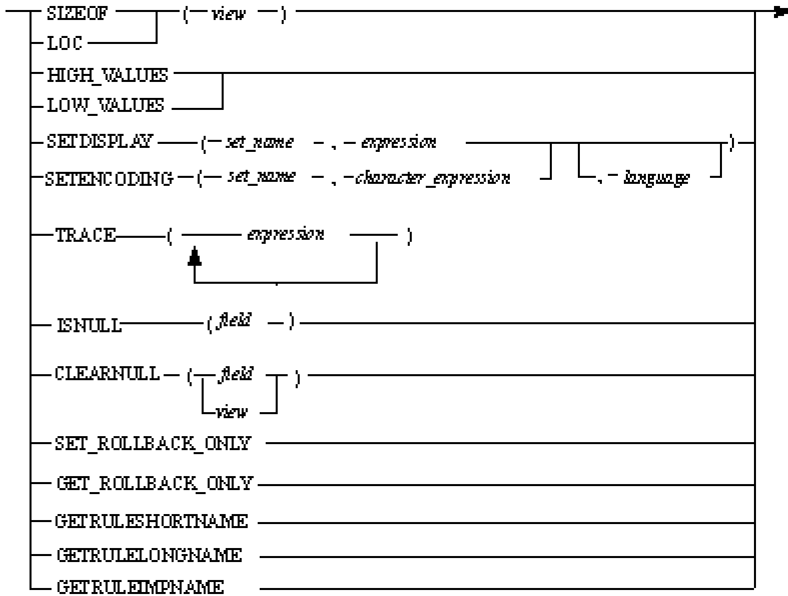


where:

- character_expression---see Character Expressions.
- num_expression---see Numeric Expressions.

Support Functions Syntax

Refer to the Support Functions for more information.

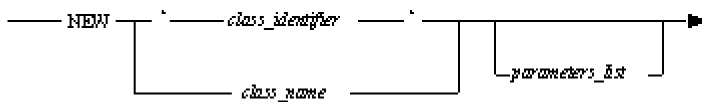


where:

- *character_expression* ---see Character Expressions.
- *expression* ---see Expression Syntax.
- *view* ---see View.

Syntax for Creating a New Object Instance in Java

Refer to the Creating a New Object Instance in Java for more information.



where:

- *parameters_list* is the list of object constructor parameters included in round brackets; if constructor has no parameters then empty brackets must be omitted.

Syntax for Dynamically-Set View Functions in Java

OCCURS Syntax

Refer to the OCCURS for more information.

where:

- *view* is any view.

APPEND Syntax

Refer to the APPEND for more information.

APPEND (target view , source view , number_of_occurs_to_process)

where:

- *target_view* must be an occurring view.
- *source_view* is any view.
- *number of occurs to process* parameter specifies how many items are taken from the *source_view* .

RESIZE Syntax

Refer to the RESIZE for more information.

RESIZE (target_view , new_size , from_position)

where:

- *target_view* is an occurring view.
- *new_size* specifies the new size of the *target_view* .
- *from_position* specifies the starting position to apply the *new_size* within the *target_view* .

DELETE Syntax

Refer to the DELETE for more information.

DELETE (target_view , from_position , number)

where:

- *target_view* is an occurring view.
- *from_position* specifies the starting position to delete.
- *number* specifies the number of occurrences to delete.

INSERT Syntax

Refer to the INSERT for more information.

INSERT (target_view , from_position , source_view , number_of_occurs_to_process)

where:

- *target_view* is an occurring view.
- *source_view* is any view.
- *from_position* specifies the position to insert.
- *number_of_occurs_to_process* specifies how many items are taken from *source_view* .
- number of occurrences to delete.

REPLACE Syntax

Refer to the REPLACE for more information.

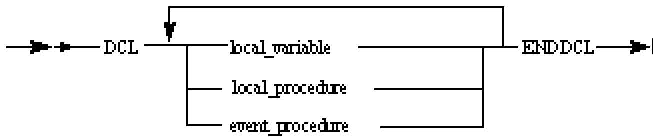
REPLACE (target_view , from_position , source_view , number_of_occurs_to_process)

where:

- *target_view* is an occurring view.
- *source_view* is any view.
- *from_position* specifies the starting position to replace.
- *number_of_occurs_to_process* specifies how many items are taken from *source_view* .

Declaration Syntax

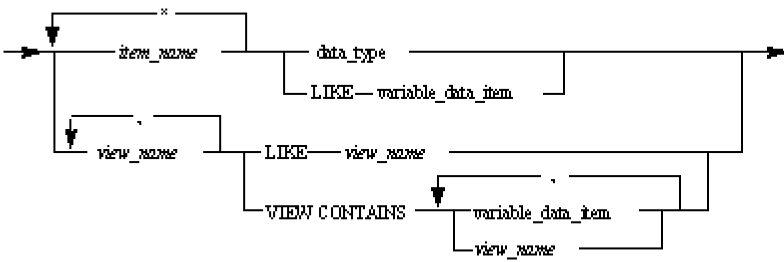
The name and data type of a variable or a procedure need to be declared before it can be used. Refer to the Declarations for more information.



Local Variable Declaration Syntax

Refer to the Local Variable Declaration for more information.

where local_variable is:

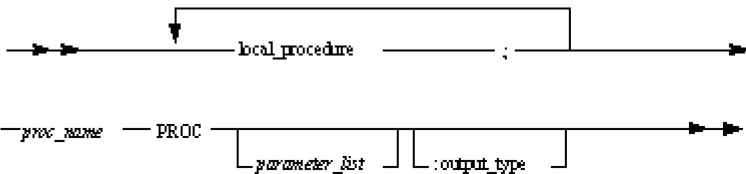


where:

- data_type---see Data Types.
- variable_data_item---see Variable Data Item.

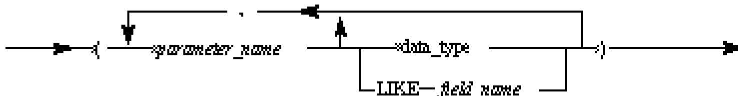
Local Procedure Declaration Syntax

Refer to the Local Procedure Declaration for more information.



where:

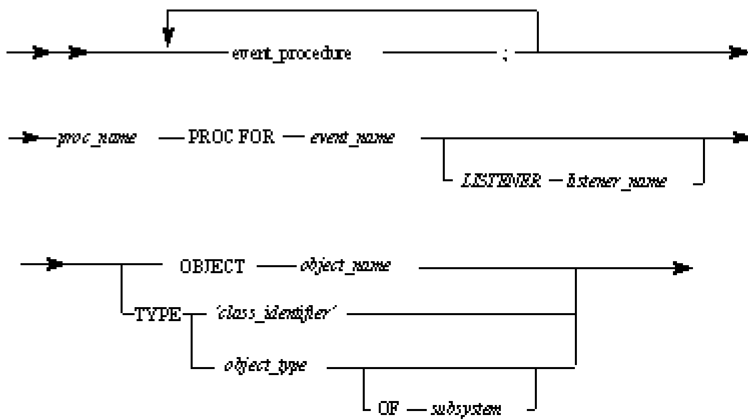
- *proc_name* is the name of a procedure to be declared.
- *parameter_list* is:



- data_type---see Data Types.

Event Procedure Declaration Syntax

Refer to the Event Procedure Declaration for more information.



where:

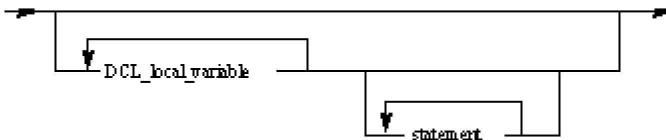
- *proc_name* is the name of a procedure to be declared.
- *event_name* is the name of the declared object event.
- *listener_name* is the name of the interface that implements event triggering (Java only).
- *object_name* can of any of the following:
- The system identifier (HPSID) of the object.
- The alias of the object---see Alias.
- A pointer to the object---see Object Data Types.
- *class_identifier* is a string that identified the class. It might be CLSID or OLE objects or fully qualified class name for Java classes. The identification string is considered case-sensitive.
- *object_type* is the type of the object whose events the procedure receives---see Object Types.
- *subsystem* is the group that the object belongs to. The following are supported:
- GUI_KERNEL, the set of window controls supplied with AppBuilder.
- JAVABEANS, for any Java class.

Common Procedure Syntax

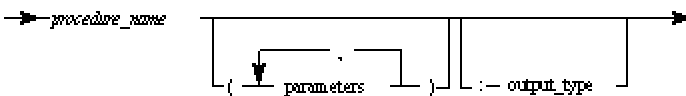
Refer to the Common Procedure for more information.

PROC — common_procedure — proc_statements — ENDPROC

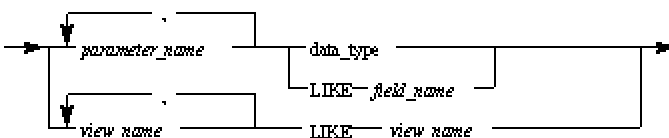
where proc_statements are:



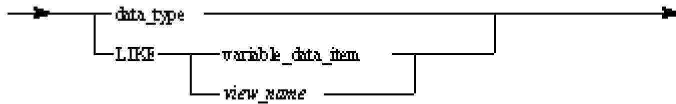
where common_procedure is:



where *parameters* can be:



where *output_type* can be:

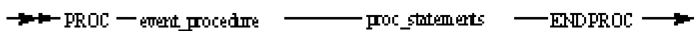


where:

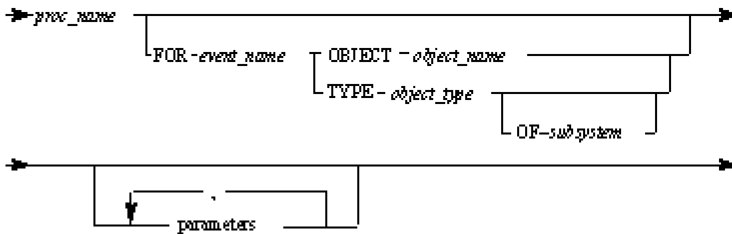
- data_type---see Data Types.
- DCL_local_variable---see Local Variable Declaration.
- variable_data_item---see Variable Data Item. Note that OBJECT array cannot be a parameter.
- statement is Any Rules Language statement, except procedure declaration.

Event Procedure Syntax

Refer to the Event Handling Procedure for more information.



where event_procedure is:

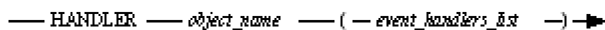


where:

- *object_name* can be any of the following:
- The HPSID of the object
- The alias of the object---see Alias.
- A pointer to the object---see Alias.
- data_type---see Data Types.
- variable_data_item---see Variable Data Item.
- parameters---see Common Procedure.
- proc_statements---see Common Procedure.
- *object_type* is the type of object whose event(s) the procedure receives.
- *subsystem* is the group to which the object pointed to belongs.

Event Handler Syntax in Java

Refer to the Event Handler Statement in Java for more information.



where:

- *object_name* is the object variable.
- *event_handlers_list* is a list of event handlers that are delimited with commas.

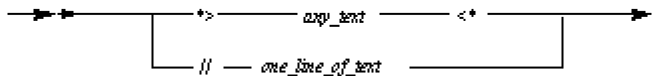
Control Statements Syntax

Refer to the Control Statements for more information about the control statements. The following control statements syntax drawings are available in this section:

- Comment Statement Syntax
- ObjectSpeak Statement Syntax
- Object Method Call Syntax
- File (Database) Access Statement Syntax
- Post Event Statement Syntax
- Compiler Pragmatic Statement Syntax

Comment Statement Syntax

Refer to the Comment Statement for more information.

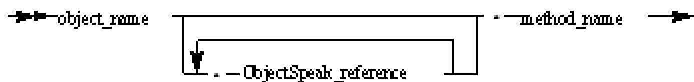


where:

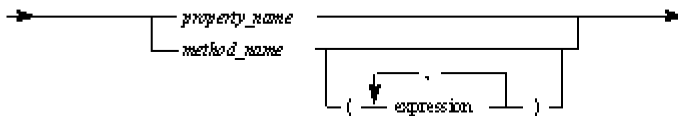
- *any_text* is any possible character sequence, including line breaks.
- *one_line_of_text* is any character sequence limited to one line (without any line breaks).

ObjectSpeak Statement Syntax

Refer to the ObjectSpeak Statement for more information.



where ObjectSpeak_reference is:



where object_name can be:

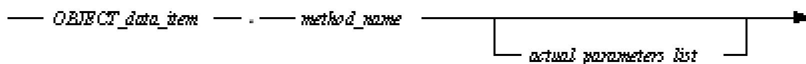
- The system identifier (HPSID) of the object
- The alias of the object---see Alias for information about alias of an object.
- An object---see Object Data Types for information about an object.
- An array---see Array Object for information about an array.

where:

- expression---see Expression Syntax.

Object Method Call Syntax

Refer to the Object Method Call for more information.

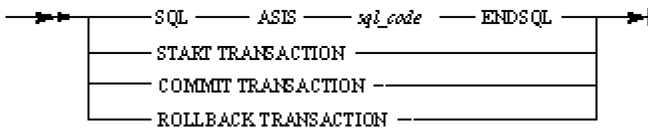


where:

- *actual_parameters_list* is a list of actual parameters delimited with commas and enclosed in parentheses. If the list is empty, parentheses can be omitted. If a method does not have parameters then empty parentheses (()) can be written. For more information, see:
- Object Method Call in C
- Object Method Call in Java

File (Database) Access Statement Syntax

Refer to the File (Database) Access Statements for more information.

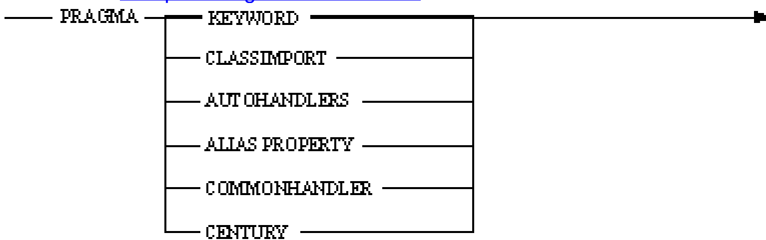


Post Event Statement Syntax

Refer to the Post Event Statement for more information.

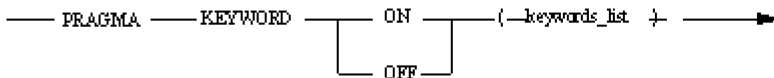
Compiler Pragmatic Statement Syntax

Refer to the Compiler Pragmatic Statements for more information.



PRAGMA KEYWORD Syntax

Refer to the PRAGMA KEYWORD for more information.

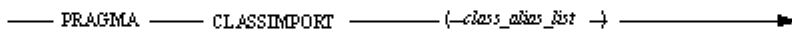


where:

- keywords_list is the parameters list of keywords to switch on or off. Separate individual keywords using commas (spaces are ignored) and place the entire list in parentheses. The PRAGMA KEYWORD clause is *not* case-sensitive, so keywords can be lower or uppercase.

PRAGMA CLASSIMPORT Syntax in Java

Refer to the PRAGMA CLASSIMPORT in Java for more information.



where:

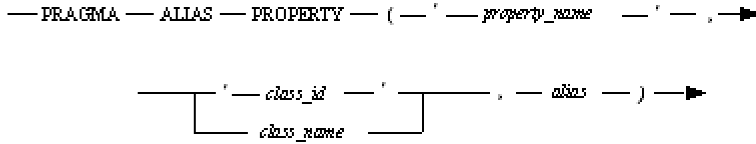
- *class_alias_list* is a list of pairs that consist of Java class name and alias to import. Separate the class name and alias using a comma (spaces are ignored), and place the entire list in parentheses. The PRAGMA CLASSIMPORT clause is case-sensitive, so note the exact capitalization of the Java class name.

PRAGMA AUTOHANDLERS Syntax in Java

Refer to the PRAGMA AUTOHANDLERS in Java for more information.

PRAGMA ALIAS PROPERTY Syntax in Java

Refer to the PRAGMA ALIAS PROPERTY in Java for more information.

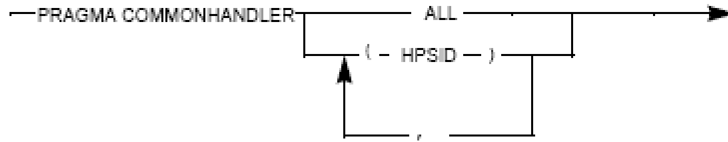


where:

- *property_name* is the case-sensitive name of a property and the alias for which it is defined.
- *class_id* is a string that identifies the implementation of the class. It might be CLSID for OLE objects or the full Java class name for Java classes. The identification string is considered to be case-sensitive.
- *class_name* is the class name used in a rule's code. It is not case-sensitive.
- *alias* is the valid Rules identifier – alias for a method. This alias can be used in Rules code instead of the method's name.

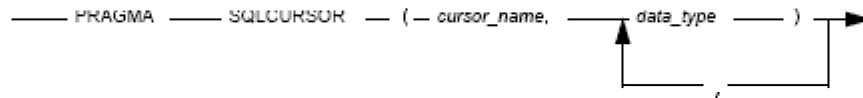
PRAGMA COMMONHANDLER Syntax in Java

Refer to the PRAGMA COMMONHANDLER in Java for more information.



PRAGMA SQLCURSOR Syntax in Java

Refer to the PRAGMA SQLCURSOR in Java for more information.



where

- *cursor_name* is any valid identifier.
- *data_type* is any primitive data type (any data type except views or objects)---see Data Types.

PRAGMA CENTURY Syntax for OpenCOBOL

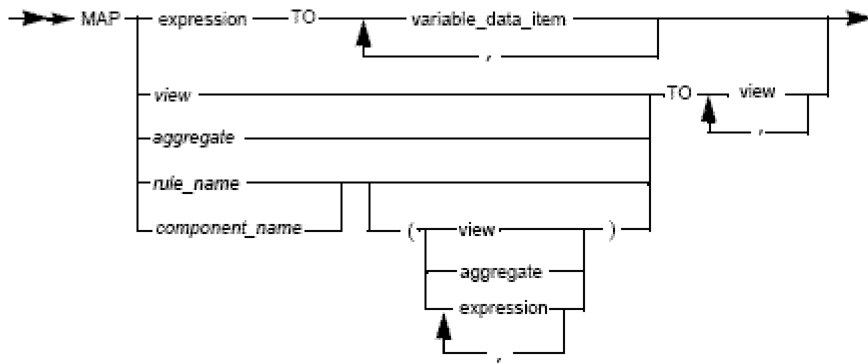
Refer to the PRAGMA CENTURY for OpenCOBOL for more information.

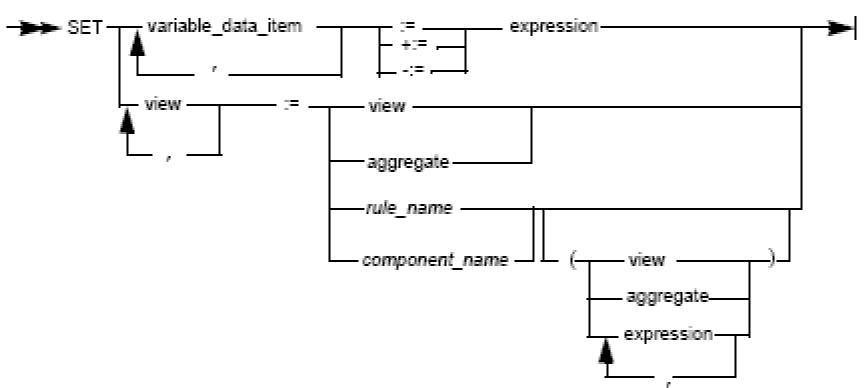
where

- *string_literal* is any character literal containing one or two digits.

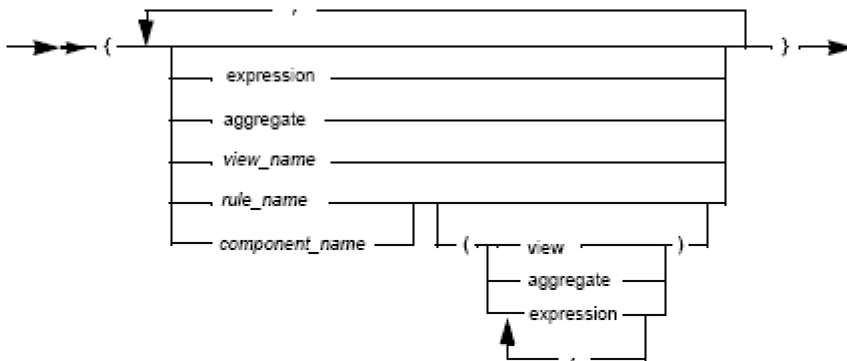
Assignment Statement Syntax

Refer to the Assignment Statements for more information.





Aggregate Syntax

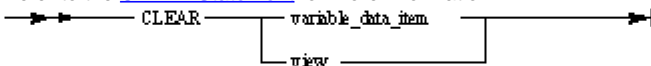


where:

- expression---see Numeric Expressions.
- variable_data_item---see Variable Data Item.
- view---see View.

CLEAR Statement Syntax

Refer to the CLEAR Statement for more information.

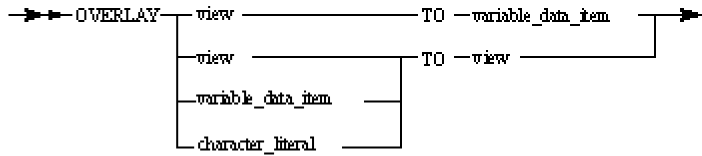


where:

- variable_data_item---see Variable Data Item.
- view---see View.

OVERLAY Statement Syntax

Refer to the OVERLAY Statement for more information.



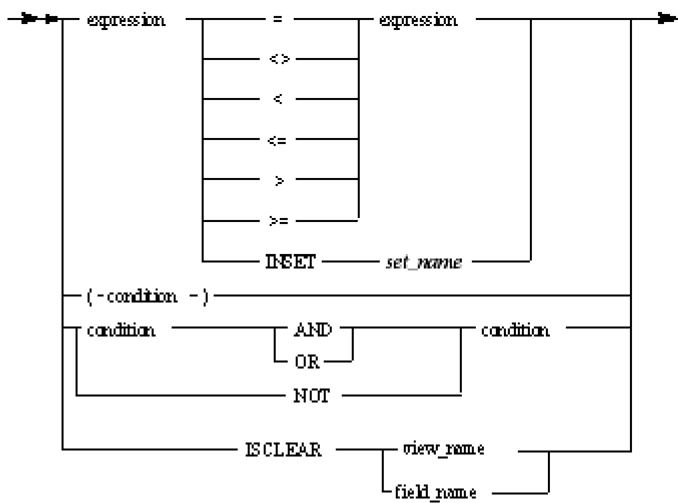
where:

- variable_data_item---see Variable Data Item.
- view---see View.
- character_literal is a Character literal---see Character Value.

Condition Operators Syntax

A condition is an expression that evaluates to either true or false. Refer to the Condition Operators for more information.

Conditions Syntax



where:

- expression---see Expression Syntax.

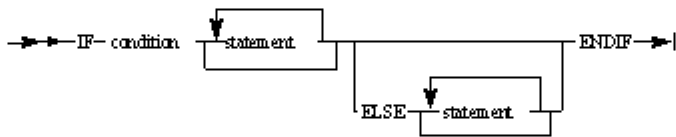
Condition Statements Syntax

Condition statements direct processing control within a rule to one group of statements or another depending on the value of a condition. Refer to the Condition Statements for more information about the condition statements. The following condition statements syntax drawings are available in this section:

- IF Statement Syntax
- CASEOF Statement Syntax
- DO Statement Syntax

IF Statement Syntax

Refer to the IF Statement for more information.

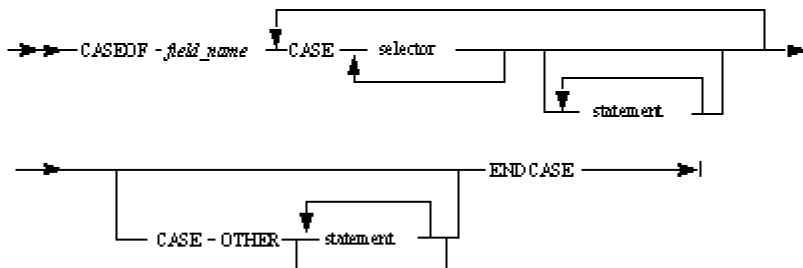


where:

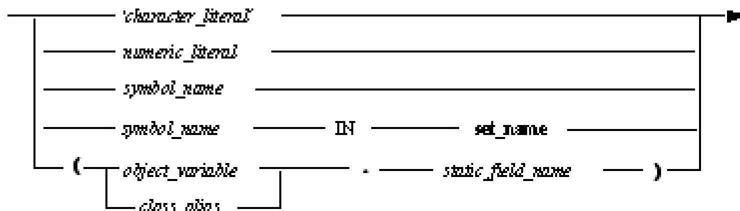
- condition---see Condition Operators.
- statement is any Rules Language statement, except a declarative statement.

CASEOF Statement Syntax

Refer to the CASEOF Statement for more information.



where selector has the following form:

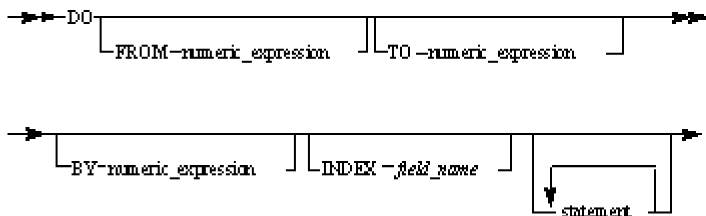


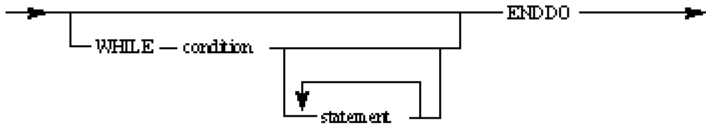
The last form of selector is available only in Java.
where:

- *symbol*--- see Symbol.
- statement is any Rules Language statement, except a declarative statement.
- *field_name* is the name of the field of a suitable type, that is, a type with constants that can appear as a selector. Allowable types are:
- Numeric
- Character

DO Statement Syntax

Refer to the DO Statement for more information.





where:

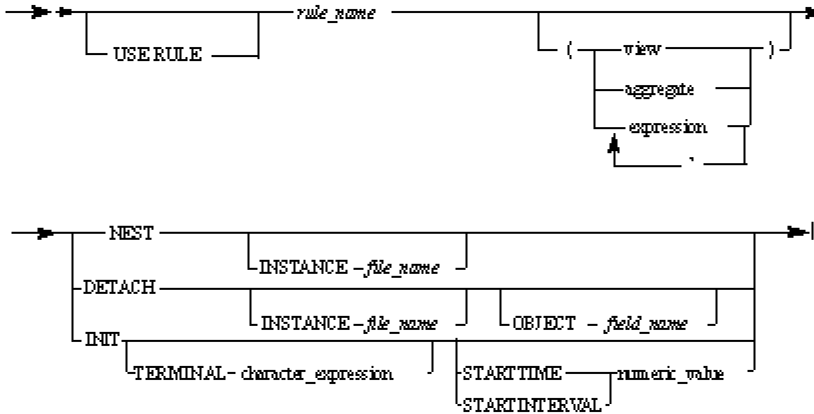
- condition---see Condition Operators.
- numeric_expression---see Numeric Expressions.
- statement is any Rules Language statement, except a declarative statement.

Transfer Statements Syntax

Transfer statements switch control of an application from one rule to another to perform another task, from a rule to a window to have the window appear on the screen, from a rule to a report to print the report, or from a rule to an internal procedure. Refer to the Transfer Statements for more information about the transfer statements. The following transfer statements syntax drawings are available in this section:

USE Statement Syntax

Refer to the USE Statements for more information.

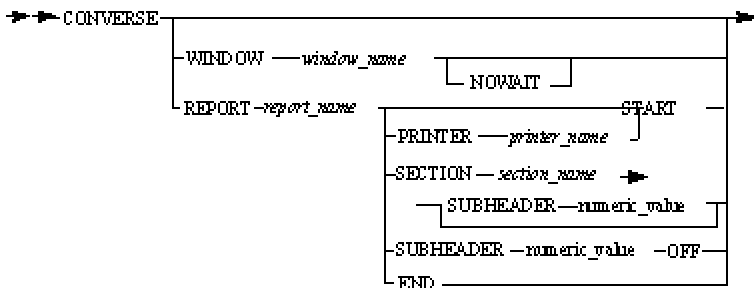


where:

- character_expression---see Character Expressions.
- expression---see Numeric Expressions.
- numeric_value---see Numeric Value.
- view---see View.

CONVERSE Statement Syntax

Refer to the CONVERSE Statements for more information.



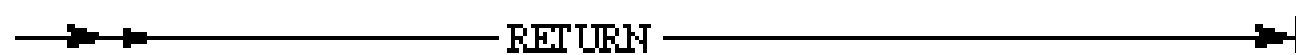
where:

- numeric_value---see Numeric Value.
- *printer_name* is a character value containing the printer name.---see Character Value.
- *report_name* is the name of the report that belongs to the current rule.

- *section_name* is the name of the section that belongs to the report (report_name).
- START is optional if the report has no sections attached.

RETURN Statement Syntax

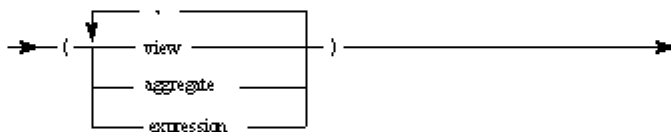
Refer to the RETURN Statement for more information.



PERFORM Statement Syntax

Refer to the PERFORM Statement for more information.

where parameter_list can be:

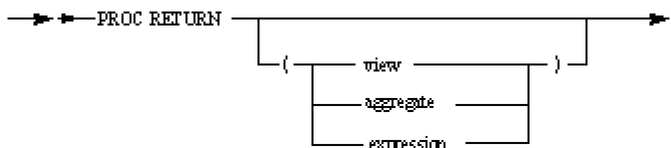


where:

- expression---see Expressions and Conditions.
- view---see View.

PROC RETURN Statement Syntax

Refer to the PROC RETURN Statement for more information.



where:

- expression is a valid expression---see Expression Syntax.
- view is a valid view---see View.

Macro Statements Syntax

Refer to the Macros for more information about the macro statements. The following control statements syntax drawings are available in this section:

- Defining Macros Syntax
- Undefining Macros Syntax
- Changing Quotes Syntax
- Syntax for Evaluating if a Macro Exists
- Syntax for Comparing Values with Macros
- Syntax for Conditional Translation with Macros
- Syntax for Including Files with Macros
- Syntax for Exiting from Translation with Macros
- Syntax for String Functions with Macros
- Syntax for Arithmetic Macros

Defining Macros Syntax

Refer to the Defining Macros for more information.

```
CG_DEFINE( macro_name [, string] )
```

where:

- *macro_name* is any sequence of letters, digits, and the underscore character (_) where the first character is not a digit. Macro names are case-sensitive, for example, "INIT" represents a different macro than "init." See Case-sensitivity for exceptions. Macro names cannot contain DBCS characters. If DBCS characters are used, an error is generated during the Rule preparation.
- *string* is any sequence of characters allowed in the Rules Language. The replacement string is not enclosed in quotation marks. (If quotation marks are included, they are part of the replacement string and are included when the replacement string is substituted for the macro name.)

Undefining Macros Syntax

Refer to the Undefining a Macro for more information.

```
CG_UNDEFINE( macro_name )
```

Changing Quotes Syntax

Refer to the Changing the Quote Characters in Macros for more information.

```
CG_CHANGEQUOTE( open, close )
```

where:

- *open* is the string to start the quotes.
- *close* is the string to end the quotes.

Syntax for Evaluating if a Macro Exists

Refer to the Evaluating if a Macro Exists for more information.

```
CG_IFDEF( macro_name , string [, string] )
```

where:

- *string* is any sequence of characters allowed in the Rules Language.

Syntax for Comparing Values with Macros

Refer to the Comparing Values for more information.

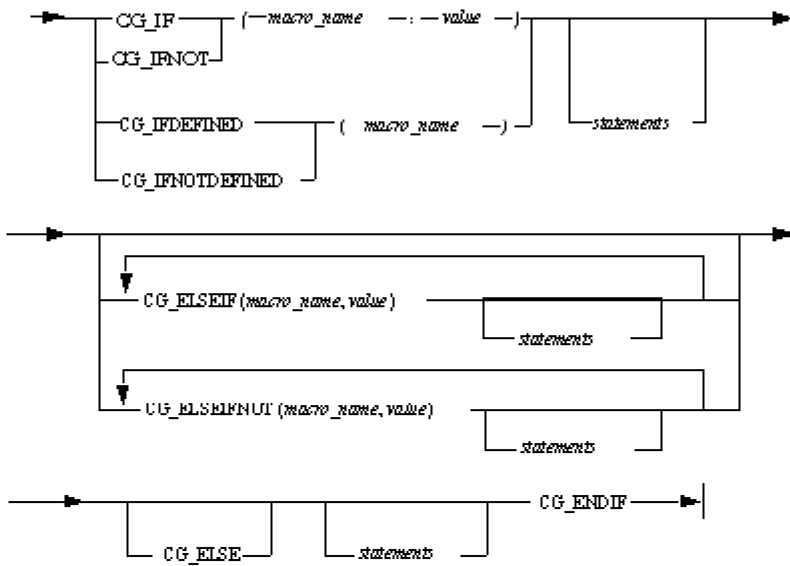
```
CG_IFELSE( value1 , value2 , string [, string ...] )
```

where:

- *value1* is the first value used in the comparison, and is typically a macro name.
- *value2* is the second value used in the comparison.
- *string* is any sequence of characters allowed in the Rules Language.

Syntax for Conditional Translation with Macros

Refer to the Using Conditional Translation for more information.



where:

- *macro_name* is any macro name.
- *value* is any string that could be assigned to *macro_name.*
- *statements* are any Rules Language statements.

Syntax for Including Files with Macros

Refer to the Including Files for more information.

where:

- *File_name* is the string specifying a file name.

Syntax for Exiting from Translation with Macros

Refer to the Exiting from Translation for more information.

where:

- *Return code* is an integer number.

Syntax for String Functions with Macros

CG_LEN Syntax

Refer to the CG_LEN for more information.

CG_INDEX Syntax

Refer to the CG_INDEX for more information.

```
►►─CG_INDEX( ─string ,─ substring ─)──────────────►◄
```

CG_SUBSTR Syntax

Refer to the CG_SUBSTR for more information.

```
►►─CG_SUBSTR( ─string ,─ from ┬──────────────┬ )──────►◄
                              └ , ─ length ──┘
```

Syntax for Arithmetic Macros

CG_INCR and CG_DECR Syntax

Refer to the CG_INCR and CG_DECR for more information.

```
►►─CG_INCR( ─number ─)────────────────────────────►◄

►►─CG_DECR( ─number ─)────────────────────────────►◄
```

CG_EVAL Syntax

Refer to the CG_EVAL for more information.

```
►►─CG_EVAL( ─expression ┬──────────────────────────┬ )──►◄
                        └ ,─radix ┬───────────────┤
                                  └ ,─ width ─────┘
```

Where:

- *expression* can contain various operators, as shown in the following table in decreasing order of precedence:

Operators Used in Expressions

Operator	Definition
-	Unary minus
**	Exponentiation
* / %	Multiplication, division and modulo
+ -	Addition and subtraction
<< >>	Shift left or right
== != > >= < <=	Relational operators
!	Logical negation
~	Bitwise negation
&	Bitwise and
^	Bitwise exclusive-or
/	Bitwise or
&&	Logical and

//	Logical or