2. THE RULE LANGUAGE

This chapter describes the syntax and semantics of the rule language.

2.1 Language Introduction

A rule in LOOPS describes actions to be taken when specified conditions are satisfied. A rule has three major parts called the *left hand side* (LHS) for describing the conditions, the *right hand side* (RHS) for describing the actions, and the *meta-description* (MD) for describing the rule itself. In the simplest case without a meta-description, there are two equivalent syntactic forms:

LHS **->** *RHS***;**

IF *LHS* **THEN** *RHS***;**

The **If** and **Then** tokens are recognized in several combinations of upper and lower case letters. The syntax for LHSs and RHSs is given below. In addition, a rule can have no conditions (meaning always perform the actions) as follows:

-> *RHS***;**

if T then *RHS***;**

Rules can be preceded by a meta-description in braces as in:

{*MD***}** *LHS* **->** *RHS***;**

{*MD***} If** *LHS* **Then** *RHS***;**

{*MD***}** *RHS***;**

Examples of meta-information include rule-specific control information, rule descriptions, audit instructions, and debugging instructions. For example, the syntax for one-shot rules shown in Section 1.5, "One-Shot Rules:"

{1} IF *condition condition* **THEN** *action* **;**

is an example of a meta-description. Another example is the use of meta-assignment statements for describing audit trails and rules. These statements are discussed in Section 1.7, "Saving an Audit Trail of Rule Invocation."

LHS Syntax: The clauses on the LHS of a rule are evaluated in order from left to right to determine whether the LHS is satisfied. If they are all satisfied, then the rule is satisfied. For example:

A B C+D (Prime D) -> *RHS*;

In this rule, there are four clauses on the LHS. If the values of some of the clauses are **NIL** during evaluation, the remaining clauses are not evaluated. For example, if **A** is non-**NIL** but **B** is **NIL**, then the LHS is not satisfied and **C+D** will not be evaluated.

RHS Syntax: The RHS of a rule consists of actions to be performed if the LHS of the rule is satisfied. These actions are evaluated in order from left to right. Actions can be the invocation of RuleSets, the sending of LOOPS messages, Interlisp function calls, variables, or special termination actions.

RuleSets always return a value. The value returned by a RuleSet is the value of the last rule that was executed. Rules can have multiple actions on the right hand side. Unless there is a **Stop** statement or transfer call as described later, the value of a rule is the value of the last action. When a rule has no actions on its RHS, it returns **NIL** as its value.

Comments: Comments can be inserted between rules in the RuleSet. They are enclosed in parentheses with an asterisk for the first character as follows:

(* This is a comment)

2.2 Kinds of Variables

LOOPS distinguishes the following kinds of variables:

RuleSet arguments: All RuleSets have the variable **self** as their workspace. References to **self** can often be elided in the RuleSet syntax. For example, the expression **self.Print** means to send a **Print** message to **self**. This expression can be shortened to **.Print** . Other arguments can be defined for RuleSets. These are declared in an **Args:** declaration.

Instance variables: All RuleSets use a LOOPS object for their workSpace. In the LHS and RHS of a rule, the first interpretation tried for an undeclared literal is as an instance variable in the work space. Instance variables can be indicated unambiguously by preceding them with a colon, (e.g., **:***varName* or *obj***:***varName*).

Class variables: Literals can be used to refer to class variables of LOOPS objects. These variables must be preceded by a double colon in the rule language, (e.g., **::***classVarName* or *obj***::***classVarName*).

Temporary variables: Literals can also be used to refer to temporary variables allocated for a specific invocation of a RuleSet. These variables are initialized to **NIL** when a RuleSet is invoked. Temporary variables are declared in the **Temporary Vars** declaration in a RuleSet.

Audit record variables: Literals can also be used to refer to instance variables of audit records created by rules. These literals are used only in *meta-assignment* statements in the MD part of a rule. They are used to describe the information saved in audit records, which can be created as a side-effect of rule execution. These variables are ignored if a RuleSet is not compiled in *audit* mode. Undeclared variables appearing on the left side of assignment statements in the MD part of a rule are treated as

audit record variables by default. These variables are declared indirectly -- they are the instance variables of the class declared as the *Audit Class* of the RuleSet.

Interlisp variables: Literals can also be used to refer to Interlisp variables during the invocation of a RuleSet. These variables can be global to the Interlisp environment, or are bound in some calling function. Interlisp variables can be used when procedure-oriented and rule-oriented programs are intermixed. Interlisp variables must be preceded by a backSlash in the syntax of the rule language (e.g., *lispVarName*).

Reserved Words: The following literals are treated as *read-only* variables with special interpretations:

self [Variable]

The current work space.

rs [Variable]

The current RuleSet.

caller [Variable]

The RuleSet that invoked the current RuleSet, or **NIL** if invoked otherwise.

ruleApplied [Variable]

Set to **T** if some rule was applied in this cycle. (For use only in while-conditions).

The following reserved words are intended mainly for use in creating audit trails:

ruleObject [Variable]

Variable bound to the object representing the rule itself.

ruleNumber [Variable]

Variable bound to the sequence number of the rule in a RuleSet.

ruleLabel [Variable]

Variable bound to the label of a rule or **NIL**.

reasons [Variable]

Variable bound a list of audit records supporting the instance variables mentioned on the LHS of the rule. (Computed at run time.)

auditObject [Variable]

Variable bound to the object to which the reason record will be attached. (Computed at run time.)

auditVarName [Variable]

Variable bound to the name of the variable on which the reason will be attached as a property.

Other Literals: As described later, literals can also refer to Interlisp functions, LOOPS objects, and message selectors. They can also be used in strings and quoted constants.

The determination of the meaning of a literal is done at compile time using the declarations and syntax of RuleSets. The characters used in literals are limited to alphabetic characters and numbers. The first character of a literal must be alphabetic.

The syntax of literals also includes a compact notation for sending unary messages and for accessing instance variables of LOOPS objects. This notation uses *compound literals*. A compound literal is a literal composed of multiple parts separated by a periods, colons, and commas.

2.3 Rule Forms

Quoted Constants: The quote sign is used to indicate constant literals:

a b=3 c='open d=f e='(This is a quoted expression) -> ...

In this example, the LHS is satisfied if **a** is non-**NIL**, and the value of **b** is 3, and the value of **c** is exactly the atom **open**, the value of **d** is the same as the value of **f**, and the value of **e** is the list **(This is a quoted expression)**.

Strings: The double quote sign is used to indicate string constants:

IF a b=3 c='open d=f e=="This is a string"
THEN (WRITE "Begin configuration task") ... ;

In this example, the LHS is satisfied if **a** is non-**NIL**, and the value of **b** is 3, and the value of **c** is exactly the atom **open**, the value of **d** is the same as the value of **f**, and the value of **e** equal to the string **"This is a string"**.

Interlisp Constants: The literals **T** and **NIL** are interpreted as the Interlisp constants of the same name.

a (Foo x NIL b) -> x_T ...;

In this example, the function **Foo** is called with the arguments **x**, **NIL**, and **b**. Then the variable **x** is set to **T**.

2.4 Infix Operators and Brackets

To enhance the readability of rules, a few infix operators are provided. The following are infix binary operators in the rule syntax:

+ [Rule Infix Operator]

Addition.

++ [Rule Infix Operator]

Addition modulo 4.

- [Rule Infix Operator]

Subtraction.

-- [Rule Infix Operator]

Subtraction modulo 4.

***** [Rule Infix Operator]

Multiplication.

/ [Rule Infix Operator]

Division.

> [Rule Infix Operator]

Greater than.

< [Rule Infix Operator]

Less than.

>= [Rule Infix Operator]

Greater than or equal.

<= [Rule Infix Operator]

Less than or equal.

= [Rule Infix Operator]

EQ -- simple form of equals. Works for atoms, objects, and small integers.

~= [Rule Infix Operator]

NEQ. (Not **EQ**.)

== [Rule Infix Operator]

EQUAL -- long form of equals.

<< [Rule Infix Operator]

Member of a list. (**FMEMB**)

In addition, the rule syntax provides two unary operators as follows:

- [Rule Unary Operator]

Minus.

~ [Rule Unary Operator]

Not.

The precedence of operators in rule syntax follows the usual convention of programming languages. For example

1+5*3 = 16

and

[3 < 2 + 4] = T

Brackets can be used to control the order of evaluation:

[1+5]*3 = 18

Ambiguity of the minus sign: Whenever there is an ambiguity about the interpretation of a minus sign as a unary or binary operator, the rule syntax interprets it as a binary minus. For example

a-b c d -e [-f] (g -h) (_ ($ Foo) Move -j) -> ...

In this example, the first and second minus signs are both treated as binary subtraction statements. That is, the first three clauses are (1) **a-b**, (2) **c** and (3) **d-e**. Because the rule syntax allows arbitary spacing between symbols and there is no syntax to separate clauses on the LHS of a rule, the interpretation of "**d -e**" is as a single clause (with the subtraction) instead of two clauses. To force the interpretation as a unary minus operator, one must use brackets as illustrated in the next clause. In this clause, the minus sign in the clause **[-f]** is treated as a unary minus because of the brackets. The minus sign in the function call **(g -h)** is treated as unary because there is no preceding argument. Similarly, the **-j** in the message expression is treated as unary because there is no preceding argument.

2.5 Interlisp Functions and Message Sending

Calls to Interlisp functions are parenthesized with the function name as the first literal after the left parenthesis. Each expression after the function name is treated as an argument to the function. For example:

a (Prime b) [a -b] -> c (Display b c+4 (Cursor x y) 2) ;

In this example, **Prime**, **Display**, and **Cursor** are interpreted as the names of Interlisp functions. Since the expression **[a -b]** is surrounded by brackets instead of parentheses, it is recognized as meaning **a** minus **b** as opposed to a call to the function **a** with the argument minus **b**. In the example above, the call to the Interlisp function **Display** has four arguments: **b**, **c+4**, the value of the function call **(Cursor x y)**, and **2**.

The use of Interlisp functions is usually outside the spirit of the rule language. However, it enables the use of Boolean expressions on the LHS beyond simple conjunctions. For example:

a (OR (NOT b) x y) z -> ... ;

LOOPS Objects and Message Sending: LOOPS classes and other named objects can be referenced by using the dollar notation. The sending of LOOPS messages is indicated by using a left arrow. For example:

IF cell_(_ ($ LowCell) Occupied? 'Heavy)
THEN (_ cell Move 3 'North);

In the LHS, an **Occupied?** message is sent to the object named **LowCell**. In the message expression on the RHS, there is no dollar sign preceding **cell**. Hence, the message is sent to the object that is the value of the variable **cell**.

For unary messages (i.e., messages with only the selector specified and the implicit argument **self**), a more compact notation is available as described selow.

Unary Message Sending: When a period is used as the separator in a compound literal, it indicates that a unary message is to be sent to an object. (We will alternatively refer to a period as a *dot*.) For example:

tile.Type='BlueGreenCross command.Type='Slide4 -> ... ;

In this example, the object to receive the unary message **Type** is referenced indirectly through the **tile** instance variable in the work space. The left literal is the variable **tile** and its value must be a LOOPS object at execution time. The right literal must be a method selector for that object.

The dot notation can be combined with the dollar notation to send unary messages to named LOOPS objects. For example,

$Tile.Type='BlueGreenCross ...

In this example, a unary **Type** message is sent to the LOOPS object whose name is **Tile**.

The dot notation can also be used to send a message to the work space of the RuleSet, that is, **self**. For example, the rule

IF scale>7 THEN .DisplayLarge;

would cause a **DisplayLarge** message to be sent to **self**. This is an abbreviation for

IF scale>7 THEN self.DisplayLarge;

2.6 Variables and Properties

When a single colon (:) is used in a literal, it indicates access to an instance variable of an object. For example:

tile:type='BlueGreenCross command:type=Slide4 -> ... ;

In this example, access to the LOOPS object is indirect in that it is referenced through an instance variable of the work space. The left literal is the variable **tile**, and its value must be a LOOPS object when the rule is executed. The right literal **type** must be the name of an instance variable of that object. The compound literal **tile:type** refers to the value of the **type** instance variable of the object in the instance variable **tile**.

The colon notation can be combined with the dollar notation to access a variable in a named LOOPS object. For example,

$TopTile:type='BlueGreenCross ...

refers to the **type** variable of the object whose LOOPS name is **TopTile**.

A double colon notation (::) is provided for accessing class variables. For example

truck::MaxGas<45 ::ValueAdded>600 -> ... ;

In this example, **MaxGas** is a class variable of the object bound to **truck**. **ValueAdded** is a class variable of **self**.

A colon-comma notation (:,) is provided for accessing property values of class and instance variables. For example

wire:,capacitance>5 wire:voltage:,support='simulation -> ...

In the first clause, **wire** is an instance variable of the work space and **capacitance** is a property of that variable. The interpretation of the second clause is left to right as usual: (1) the object that is the value of the variable **wire** is retrieved, and (2) the **support** property of the **voltage** variable of that object is retrieved. For properties of class variables

::Wire:,capacitance>5 node::Voltage:,support='simulation -> ...

In the first clause, **wire** is a class variable of the work space and **capacitance** is a property of that variable. In the second clause, **node** is an instance variable bound to some object. **Voltage** is a class variable of that object, and **Support** is a property of that class variable.

The property notation is illegal for ruleVars and lispVars since those variables cannot have properties.

2.7 Computing Selectors and Variable Names

The short notations for instance variables, properties, and unary messages all show the selector and variable names *as they actually appear* in the object.

*object***.***selector*
*object***:***ivName*
*object***::***cvName*
*object***:***varname***:,***propName*

(_ *object selector arg arg* **)**

For example,

apple:flavor

refers to the **flavor** instance variable of the object bound to the variable **apple**. In Interlisp terminology, this implies implicit quoting of the name of the instance variable (**flavor**).

In some applications it is desired to be able to compute the names. For this, the LOOPS rule language provides analogous notations with an added exclamation sign (!). After the exclamation sign, the interpretation of the variable being evaluated starts over again. For example

apple:!\x

refers to the same thing as **apple:flavor** if the Interlisp variable **x** is bound to **flavor**. The fact that **x** is a Lisp variable is indicated by the backslash. If **x** is an instance variable of **self** or a temporary variable, we could use the notation:

apple:!x

If **x** is a class variable of **self**, we could use the notation:

apple:!::x

All combinations are possible, including:

*object***.!***selector*
*object***.!***selector*
*object***.!::***selector*
*object***:!***ivName*
*object***::!***cvName*
*object***:!***varname***:,***propName*

(_! *object selector arg arg* **)**

2.8 Recursive Compound Literals

Multiple colons or periods can be used in a literal, For example:

a:b:c

means to (1) get the object that is the value of **a**, (2) get the object that is the value of the **b** instance variable of **a**, and finally (3) get the value of the **c** instance variable of that object.

Similarly, the notation

a.b:c

means to get the **c** variable of the object returned after sending a **b** message to the object that is the value of the variable **a**. Again, the operations are carried out left to right: (1) the object that is the value of the variable **a** is retrieved, (2) it is sent a **b** message which must return an object, and then (3) the value of the **c** variable of that object is retrieved.

Compound literal notation can be nested arbitrarily deeply.

2.9 Assignment Statements

An assignment statement using a left arrow can be used for setting all kinds of variables. For example,

x_a;

sets the value of the variable **x** to the value of **a**. The same notation works if **x** is a task variable, rule variable, class variable, temporary variable, or work space variable. The right side of an assignment statement can be an expression as in:

x_a*b + 17*(LOG d);

The assignment statement can also be used with the colon notation to set values of instance variables of objects. For example:

y:b_0 ;

In this example, first the object that is the value of **y** is computed, then the value of its instance variable **b** is set to **0**.

Properties: Assignment statements can also be used to set property values as in:

box:x:,origin_47 fact:,reason_currentSupport;

Nesting: Assignment statements can be nested as in

a_b_c:d_3;

This statement sets the values of **a**, **b**, and the **d** instance variable of **c** to **3**. The value of an assignment statement itself is the new assigned value.

2.10 Meta-Assignment Statements

Meta-assignment statements are assignment statements used for specifying rule descriptions and audit trails. These statements appear in the MD part of rules.

Audit Trails: The default interpretation of meta-assignment statements for undeclared variables is as audit trail specifications. Each meta-assignment statement specifies information to be saved in audit records when a rule is applied. In the following example from Figure 4, the audit record must have variables named **basis** and **cf**:

{(basis_Fact cf_1.)}
IF buyer:familySize>2 machine:capacity<20
THEN suitability_'Poor;

In this example, the RHS of the rule assigns the value of the work space instance variable **suitability** to **'Poor** if the conditions of the rule are satisfied. In addition, if the RuleSet was compiled in *audit* mode, then during RuleSet execution an audit record is created as a side-effect of the assignment. The audit record is attached to the **reason** property of the suitability variable. It has instance variables **basis** and **cf**.

In general, an audit description consists of a sequence of meta-assignment statements. The assignment variable on the left must be an instance variable of the audit record. The class of the audit record is declared in the *Audit Class* declaration of the RuleSet. The expression on the right is in terms of the variables accessible by the RuleSet. If the conditions of a rule are satisfied, an audit record is instantiated. Then the meta-assignment statements are evaluated in the execution context of the RuleSet and their values are put into the audit record. A separate audit record is created for each of the object variables that are set by the rule.

2.11 Push and Pop Statements

A compact notation is provided for pushing and popping values from lists. To push a new value onto a list, the notation _**+** is used:

myList_+newItem;

focus:goals_+newGoal;

To pop an item from a list, the _**-** notation is used:

item_-myList;

nextGoal_-focus:goals;

As with the assignment operator, the push and pop notation works for all kinds of variables and properties. They can be used in conjunction with infix operator **<<** for membership testing.

2.12 Invoking RuleSets

One of the ways to cause RuleSets to be executed is to invoke them from rules. This is used on the LHS of rules to express predicates in terms of RuleSets, and on the RHS of rules to express actions in terms of RuleSets. A short double-dot syntax(..) for this is provided that invokes a RuleSet on a work space:

Rs1..ws1

In this example, the RuleSet bound to the variable **Rs1** is invoked with the value of the variable **ws1** as its work space. The value of the invocation expression is the value returned by the RuleSet. The double-dot syntax can be combined with the dollar notation ($) to invoke a RuleSet by its LOOPS name, as in

$MyRules..ws1

which invokes the RuleSet object that has the LOOPS name **MyRules**.

This form of RuleSet invocation is like subroutine calling, in that it creates an implicit stack of arguments and return addresses. This feature can be used as a mechanism for *meta-control* of RuleSets as in:

IF breaker:status='Open
THEN source_$OverLoadRules..washingMachine;

IF source='NotFound
THEN $ShortCircuitRules..washingMachine;

In this example, two "meta-rules" are used to control the invocation of specialized RuleSets for diagnosing overloads or short circuits.

2.13 Transfer Calls

An important optimization in many recursive programs is the elimination of tail recursion. For example, suppose that the RuleSet A calls B, B calls C, and C calls A recursively. If the first invocation of A must do some more work after returning from B, then it is useful to save the intermediate states of each of the procedures in frames on the calling stack. For such programs, the space allocation for the stack must be enough to accommodate the maximum depth of the calls.

There is a common and special case, however, in which it is unnecessary to save more than one frame on the stack. In this case each RuleSet has no more work to do after invoking the other RuleSets, and the value of each RuleSet is the value returned by the RuleSet that it invokes. RuleSet invocation in this case amounts to the evaluation of arguments followed by a direct transfer of control. We call such invocations transfer calls.

The LOOPS rule language extends the syntax for RuleSet invocation and message sending to provide this as follows:

RS..*ws

The RuleSet **RS** is invoked on the work space **ws**. With transfer calls, RuleSet invocations can be arbitrarily deep without using proportional stack space.

2.14 Stop Statements

To provide premature terminations in the execution of a RuleSet, the Stop statement is provided.

(Stop *value***)** [RuleSet Statement]

value is the value to be returned by the RuleSet.

[This page intentionally left blank]