3.1 Slot Default Values ..... 22
3.2 Slot Default Constraints for Pattern-Matching ..... 22
3.3 Slot Value Constraint Attributes ..... 23
3.4 Implied Deftemplates ..... 23

**Section 4 - Deffacts Construct ..... 25**

**Section 5 - Defrule Construct ..... 27**
5.1 Defining Rules ..... 27
5.2 Basic Cycle Of Rule Execution ..... 28
5.3 Conflict Resolution Strategies ..... 29
5.3.1 Depth Strategy ..... 29
5.3.2 Breadth Strategy ..... 29
5.3.3 Simplicity Strategy ..... 29
5.3.4 Complexity Strategy ..... 30
5.3.5 LEX Strategy ..... 30
5.3.6 MEA Strategy ..... 31
5.3.7 Random Strategy ..... 31
5.4 LHS Syntax ..... 33
5.4.1 Pattern Conditional Element ..... 33
5.4.1.1 Literal Constraints ..... 34
5.4.1.2 Wildcards Single- and Multifield ..... 36
5.4.1.3 Variables Single- and Multifield ..... 38
5.4.1.4 Connective Constraints ..... 40
5.4.1.5 Predicate Constraints ..... 43
5.4.1.6 Return Value Constraints ..... 45
5.4.1.7 Pattern-Matching with Object Patterns ..... 46
5.4.1.8 Pattern-Addresses ..... 49
5.4.2 Test Conditional Element ..... 49
5.4.3 Or Conditional Element ..... 51
5.4.4 And Conditional Element ..... 52
5.4.5 Not Conditional Element ..... 53
5.4.6 Exists Conditional Element ..... 54
5.4.7 Forall Conditional Element ..... 56
5.4.8 Logical Conditional Element ..... 58
5.4.9 Automatic Addition and Reordering of LHS CEs ..... 61
5.4.9.1 Rules Without Any LHS Pattern CEs ..... 61
5.4.9.2 Test and Not CEs as the First CE of an And CE ..... 61
5.4.9.3 Test CEs Following Not CEs ..... 62
5.4.9.4 Or CEs Following Not CEs ..... 63
5.4.9.5 Notes About Pattern Addition and Reordering ..... 63
5.4.10 Declaring Rule Properties ..... 63
5.4.10.1 The Salience Rule Property ..... 64

Preview from Notesale.co.uk
Page 4 of 428

9.4 Defmessage-handler Construct ..... 103
9.4.1 Message-handler Parameters ..... 105
9.4.1.1 Active Instance Parameter ..... 105
9.4.2 Message-handler Actions ..... 106
9.4.3 Daemons ..... 108
9.4.4 Predefined System Message-handlers ..... 108
9.4.4.1 Instance Initialization ..... 108
9.4.4.2 Instance Deletion ..... 109
9.4.4.3 Instance Display ..... 110
9.4.4.4 Directly Modifying an Instance ..... 110
9.4.4.5 Modifying an Instance using Messages ..... 111
9.4.4.6 Directly Duplicating an Instance ..... 111
9.4.4.7 Duplicating an Instance using Messages ..... 112
9.4.4.8 Instance Creation ..... 112
9.5 Message Dispatch ..... 112
9.5.1 Applicability of Message-handlers ..... 113
9.5.2 Message-handler Precedence ..... 113
9.5.3 Shadowed Message-handlers ..... 114
9.5.4 Message Execution Errors ..... 114
9.5.5 Message Return Value ..... 115
9.6 Manipulating Instances ..... 115
9.6.1 Creating Instances ..... 115
9.6.1.1 Definstances Construct ..... 117
9.6.2 Reinitializing Existing Instances ..... 118
9.6.3 Reading Slots ..... 120
9.6.4 Setting Slots ..... 120
9.6.5 Deleting Instances ..... 121
9.6.6 Delayed Pattern-Matching When Manipulating Instances ..... 121
9.6.7 Modifying Instances ..... 122
9.6.7.1 Directly Modifying an Instance with Delayed Pattern-Matching ..... 122
9.6.7.2 Directly Modifying an Instance with Immediate Pattern-Matching ..... 123
9.6.7.3 Modifying an Instance using Messages with Delayed Pattern-Matching ..... 123
9.6.7.4 Modifying an Instance using Messages with Immediate Pattern-Matching . 124
9.6.8 Duplicating Instances ..... 124
9.6.8.1 Directly Duplicating an Instance with Delayed Pattern-Matching ..... 124
9.6.8.2 Directly Duplicating an Instance with Immediate Pattern-Matching ..... 125
9.6.8.3 Duplicating an Instance using Messages with Delayed Pattern-Matching ... 125
9.6.8.4 Duplicating an Instance using Messages with Immediate Pattern-Matching 126
9.7 Instance-set Queries and Distributed Actions ..... 127
9.7.1 Instance-set Definition ..... 128
9.7.2 Instance-set Determination ..... 129
9.7.3 Query Definition ..... 130
9.7.4 Distributed Action Definition ..... 131

Preview from Notesale.co.uk
Page 6 of 428

9.7.5 Scope in Instance-set Query Functions ..... 132
9.7.6 Errors during Instance-set Query Functions ..... 132
9.7.7 Halting and Returning Values from Query Functions ..... 133
9.7.8 Instance-set Query Functions ..... 133
9.7.8.1 Testing if Any Instance-set Satisfies a Query ..... 133
9.7.8.2 Determining the First Instance-set Satisfying a Query ..... 133
9.7.8.3 Determining All Instance-sets Satisfying a Query ..... 134
9.7.8.4 Executing an Action for the First Instance-set Satisfying a Query ..... 134
9.7.8.5 Executing an Action for All Instance-sets Satisfying a Query ..... 135
9.7.8.6 Executing a Delayed Action for All Instance-sets Satisfying a Query ..... 135

**Section 10 - Defmodule Construct ..... 137**
10.1 Defining Modules ..... 137
10.2 Specifying a Construct's Module ..... 138
10.3 Specifying Modules ..... 139
10.4 Importing and Exporting Constructs ..... 139
10.4.1 Exporting Constructs ..... 140
10.4.2 Importing Constructs ..... 141
10.5 Importing and Exporting Facts and Instances ..... 141
10.5.1 Specifying Instance-Names ..... 142
10.6 Modules and Rule Execution ..... 142

**Section 11 - Constraint Attributes ..... 145**
11.1 Type Attribute ..... 145
11.2 Allowed Constant Attributes ..... 146
11.3 Range Attribute ..... 147
11.4 Cardinality Attribute ..... 147
11.5 Deriving a Default Value From Constraints ..... 148
11.6 Constraint Violation Examples ..... 149

**Section 12 - Actions And Functions ..... 151**
12.1 Predicate Functions ..... 151
12.1.1 Testing For Numbers ..... 151
12.1.2 Testing For Floats ..... 151
12.1.3 Testing For Integers ..... 151
12.1.4 Testing For Strings Or Symbols ..... 152
12.1.5 Testing For Strings ..... 152
12.1.6 Testing For Symbols ..... 152
12.1.7 Testing For Even Numbers ..... 152
12.1.8 Testing For Odd Numbers ..... 152
12.1.9 Testing For Multifield Values ..... 153
12.1.10 Testing For External-Addresses ..... 153
12.1.11 Comparing for Equality ..... 153

Preview from Notesale.co.uk
Page 7 of 428

```
CLIPS> (defglobal ?*x* = 3)
CLIPS> ?*x*
3
CLIPS> red
red
CLIPS> (bind ?a 5)
5
CLIPS> (+ ?a 3)
8
CLIPS> (reset)
CLIPS> ?a
[EVALUATN1] Variable a is unbound
FALSE
CLIPS>
```

The previous example first called the addition function adding the numbers 3 and 4 to yield the result 7. A global variable ?*x* was then defined and given the value 3. The variable ?*x* was then entered at the prompt and its value of 3 was returned. Finally the constant symbol *red* was entered and was returned (since a constant evaluates to itself).

### 2.1.2 Automated Command Entry and Loading

Some operating systems allow additional arguments to be specified to a program when it begins execution. When the CLIPS executable is started under such an operating system, CLIPS can be made to automatically execute a series of commands read directly from a file or to load constructs from a file. The command-line syntax for starting CLIPS and automatically reading commands or loading constructs from a file is as follows:

Preview from Notesale.co.uk
Page 30 of 428

**Syntax**

```
clips <option>*

<option> ::= -f <filename> |
             -f2 <filename> |
             -l <filename>
```

For the **-f** option, <filename> is a file that contains CLIPS commands. If the **exit** command is included in the file, CLIPS will halt and the user is returned to the operating system after executing the commands in the file. If an **exit** command is not in the file, CLIPS will enter in its interactive state after executing the commands in the file. Commands in the file should be entered exactly as they would be interactively (i.e. opening and closing parentheses must be included and a carriage return must be at the end of the command). The **-f** command line option is equivalent to interactively entering a **batch** command as the first command to the CLIPS prompt.

The **-f2** option is similar to the **-f** option, but is equivalent to interactively entering a **batch*** command. The commands stored in <filename> are immediately executed, but the commands and their return values are not displayed as they would be for a **batch** command.

```
<lexeme> ::= <symbol> | <string>
```

A complete BNF listing for CLIPS constructs along with some commonly used replacements for non-terminal symbols are listed in appendix I.

## 2.3 BASIC PROGRAMMING ELEMENTS

CLIPS provides three basic elements for writing programs: primitive data types, functions for manipulating data, and constructs for adding to a knowledge base.

### 2.3.1 Data Types

CLIPS provides eight primitive data types for representing information. These types are **float**, **integer**, **symbol**, **string**, **external-address**, **fact-address**, **instance-name** and **instance-address**. Numeric information can be represented using floats and integers. Symbolic information can be represented using symbols and strings.

Preview from Notesale.co.uk
Page 32 of 428

A **number** consists *only* of digits (0-9), a decimal point (.), a sign (+ or -), and, optionally, an (e) for exponential notation with its corresponding sign. A number is either stored as a float or an integer. Any number consisting of an optional sign followed by only digits is stored as an **integer** (represented internally by CLIPS as a C long integer). All other numbers are stored as **floats** (represented internally by CLIPS as a C double precision float). The number of significant digits will depend on the machine implementation. Roundoff errors also may occur, again depending on the machine implementation. As with any computer language, care should be taken when comparing floating-point values to each other or comparing integers to floating-point values. Some examples of integers are

```
237          15          +12          -32
```

Some examples of floats are

```
237e3        15.09       +12.0        -32.3e-7
```

Specifically, integers use the following format:

```
<integer> ::= [+ | -] <digit>+

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Floating point numbers use the following format:

```
<float> ::= <integer> <exponent> |

            <integer> . [exponent]
```

```
                . <unsigned integer> [exponent]

                <integer> . <unsigned integer> [exponent]

<unsigned-integer> ::= <digit>+

<exponent> ::= e | E <integer>
```

A sequence of characters which does not exactly follow the format of a number is treated as a symbol (see the next paragraph).

A **symbol** in CLIPS is any sequence of characters that starts with any printable ASCII character and is followed by zero or more printable ASCII characters. When a delimiter is found, the symbol is ended. The following characters act as **delimiters**: any non-printable ASCII character (including spaces, tabs, carriage returns, and line feeds), a double quote, opening and closing parentheses "(" and ")", an ampersand "&", a vertical bar "|", a less than "<", and a tilde "~". A semicolon ";" starts a CLIPS comment (see section 2.3.3) and also acts as a delimiter. Delimiters may not be included in symbols with the exception of the "<" character which may be the first character in a symbol. In addition, a symbol may not begin with either the "?" character or the "$?" sequence of characters (although a symbol may contain these characters). These characters are reserved for variables (which are discussed later in this section). CLIPS is case sensitive (i.e. uppercase letters will match only uppercase letters). Note that numbers are a special case of symbols (i.e. they satisfy the definition of a symbol, but they are treated as a different data type). Some simple examples of symbols are

```
fOO             Hello           B7&_HI          bad_value

127A            456-93-039      @+=-%           2each
```

A **string** is a set of characters that starts with a double quote (") and is followed by zero or more printable characters. A string ends with double quotes. Double quotes may be embedded within a string by placing a backslash (\) in front of the character. A backslash may be embedded by placing two consecutive backslash characters in the string. Some examples are

```
"foo"           "a and b"       "1 number"      "a\"quote"
```

Note that the string "abcd" is not the same as the symbol *abcd*. They both contain the same characters, but are of different types. The same holds true for the instance name [abcd].

An **external-address** is the address of an external data structure returned by a function (written in a language such as C or Ada) that has been integrated with CLIPS. This data type can only be created by calling a function (i.e. it is not possible to specify an external-address by typing the value). In the basic version of CLIPS (which has no user defined external functions), it is not possible to create this data type. External-addresses are discussed in further detail in the

Preview from Notesale.co.uk
Page 33 of 428

Function calls in CLIPS use a prefix notation – the arguments to a function always appear after the function name. Function calls begin with a left parenthesis, followed by the name of the function, then the arguments to the function follow (each argument separated by one or more spaces). Arguments to a function can be primitive data types, variables, or another function call. The function call is then closed with a right parenthesis. Some examples of function calls using the addition (+) and multiplication (*) functions are shown following.

```
(+ 3 4 5)
(* 5 6.0 2)
(+ 3 (* 8 9) 4)
(* 8 (+ 3 (* 2 3 4) 9) (* 3 4))
```

While a function refers to a piece of executable code identified by a specific name, an **expression** refers to a function which has its arguments specified (which may or may not be functions calls as well). Thus the previous examples are expressions which make calls to the * and + functions.

### 2.3.3 Constructs

Several defining **constructs** appear in CLIPS: **defmodule**, **defrule**, **deffacts**, **deftemplate**, **defglobal**, **deffunction**, **defclass**, **definstances**, **defmessage-handler**, **defgeneric**, and **defmethod**. All constructs in CLIPS are surrounded by parentheses. The construct opens with a left parenthesis and closes with a right parenthesis. Defining a construct differs from calling a function primarily in effect. Typically a function call leaves the CLIPS environment unchanged (with some notable exceptions such as resetting or clearing the environment or opening a file). Defining a construct, however, is explicitly intended to alter the CLIPS environment by adding to the CLIPS knowledge base. Unlike function calls, constructs never have a return value.

Preview from Notesale.co.uk
Page 36 of 428

As with any programming language, it is highly beneficial to comment CLIPS code. All constructs (with the exception of defglobal) allow a comment directly following the construct name. Comments also can be placed within CLIPS code by using a semicolon (;). Everything from the semicolon until the next return character will be ignored by CLIPS. If the semicolon is the first character in the line, the entire line will be treated as a comment. Examples of commented code will be provided throughout the reference manual. Semicolon commented text is not saved by CLIPS when loading constructs (however, the optional comment string within a construct is saved).

## 2.4 DATA ABSTRACTION

There are three primary formats for representing information in CLIPS: facts, objects and global variables.

statement. In contrast, rules act like WHENEVER-THEN statements. The inference engine always keeps track of rules which have their conditions satisfied and thus rules can immediately be executed when they are applicable. In this sense, rules are similar to exception handlers found in languages such as Ada.

### 2.5.2 Procedural Knowledge

CLIPS also supports a procedural paradigm for representing knowledge like that of more conventional languages, such as Pascal and C. Deffunctions and generic functions allow the user to define new executable elements to CLIPS that perform a useful side-effect or return a useful value. These new functions can be called just like the built-in functions of CLIPS. Message-handlers allow the user to define the behavior of objects by specifying their response to messages. Deffunctions, generic functions and message-handlers are all procedural pieces of code specified by the user that CLIPS executes interpretively at the appropriate times. Defmodules allow a knowledge base to be partitioned.

#### 2.5.2.1 Deffunctions

Deffunctions allow you to define new functions in CLIPS directly. In previous versions of CLIPS, the only way to have user-defined functions was to write them in some external language, such as C or Ada, and then recompile and relink CLIPS with the new functions. The body of a deffunction is a series of expressions similar to the RHS of a rule that are executed in order by CLIPS when the deffunction is called. The return value of a deffunction is the value of the last expression evaluated within the deffunction. Calling a deffunction is identical to calling any other function in CLIPS. Deffunctions are covered comprehensively in Section 7.

Preview from Notesale.co.uk
Page 42 of 428

#### 2.5.2.2 Generic Functions

Generic functions are similar to deffunctions in that they can be used to define new procedural code directly in CLIPS, and they can be called like any other function. However, generic functions are much more powerful because they can be **overloaded**. A generic function will do different things depending on the types (or classes) and number of its arguments. Generic functions are comprised of multiple components called methods, where each method handles different cases of arguments for the generic function. For example, you might overload the “+” operator to do string concatenation when it is passed strings as arguments. However, the “+” operator will still perform arithmetic addition when passed numbers. There are two methods in this example: an explicit one for strings defined by the user and an implicit one which is the standard CLIPS arithmetic addition operator. The return value of a generic function is the evaluation of the last expression in the method executed. Generic functions are covered comprehensively in Section 8.

numbers as arguments, or you can define message-handlers for the NUMBER class which allow you to do it in the purely OOP fashion.

All programming elements which are not objects must be manipulated in a non-OOP utilizing function tailored for those programming elements. For example, to print a rule, you call the function **ppdefrule**; you do not send a message "print" to a rule, since it is not an object.

### 2.6.2 Primary OOP Features

There are five primary characteristics that an OOP system must possess: **abstraction**, **encapsulation**, **inheritance**, **polymorphism** and **dynamic binding**. An abstraction is a higher level, more intuitive representation for a complex concept. Encapsulation is the process whereby the implementation details of an object are masked by a well-defined external interface. Classes may be described in terms of other classes by use of inheritance. Polymorphism is the ability of different objects to respond to the same message in a specialized manner. Dynamic binding is the ability to defer the selection of which specific message-handlers will be called for a message until run-time.

Preview from Notesale.co.uk
Page 44 of 428

The definitions of new classes allows the abstraction of new data types in COOL. The slots and message-handlers of these classes describe the properties and behavior of a new group of objects.

COOL supports encapsulation by requiring message-passing for the manipulation of instances of user-defined classes. An instance cannot respond to a message for which it does not have a defined message-handler.

COOL allows the user to specify some or all of the properties and behavior of a class in terms of one or more unrelated superclasses. This process is called **multiple inheritance**. COOL uses the existing hierarchy of classes to establish a linear ordering called the **class precedence list** for a new class. Objects which are instances of this new class can inherit properties (slots) and behavior (message-handlers) from each of the classes in the class precedence list. The word precedence implies that properties and behavior of a class first in the list override conflicting definitions of a class later in the list.

One COOL object can respond to a message in a completely different way than another object; this is polymorphism. This is accomplished by attaching message-handlers with differing actions but which have the same name to the classes of these two objects respectively.

Dynamic binding is supported in that an object reference (see section 2.3.1) in a **send** function call is not bound until run-time. For example, an instance-name or variable might refer to one object at the time a message is sent and another at a later time.

conflict resolution strategies). The lex and mea strategies are provided to help in converting OPS5 programs to CLIPS.

The random strategy is useful for testing. Because this strategy randomly orders activations having the same salience, it is useful in detecting whether the execution order of rules with the same salience effects the program behavior. Before running a program with the random strategy, first seed the random number generator using the **seed** function. The same seed value can be subsequently be used if it is necessary to replicate the results of the program run.

## 5.4 LHS SYNTAX

This section describes the syntax used on the LHS of a rule. The LHS of a CLIPS rule is made up of a series of conditional elements (CEs) that must be satisfied for the rule to be placed on the agenda. There are eight types of conditional elements: **pattern** CEs, **test** CEs, **and** CEs, **or** CEs, **not** CEs, **exists** CEs, **forall** CEs, and **logical** CEs. The **pattern** CE is the most basic and commonly used conditional element. **Pattern** CEs contain constraints which are used to determine if any pattern entities (facts or instances) satisfy the pattern. The **test** CE is used to evaluate expressions as part of the pattern-matching process. The **and** CE is used to specify that an entire group of CEs must all be satisfied. The **or** CE is used to specify that only one of a group of CEs must be satisfied. The **not** CE is used to specify that a CE must not be satisfied. The **exists** CE is used to test for the occurrence of at least one partial match for a set of CEs. The **forall** CE is used to test that a set of CEs is satisfied for every partial match of a specified CE. Finally, the **logical** CE allows assertions of facts and the creation of instances on the RHS of a rule to be logically dependent upon pattern entities matching patterns on the LHS of a rule (truth maintenance).

Preview from Notesale.co.uk
Page 59 of 428

**Syntax**

```
<conditional-element> ::= <pattern-CE> |
                          <assigned-pattern-CE> |
                          <not-CE> |
                          <and-CE> |
                          <or-CE> |
                          <logical-CE> |
                          <test-CE> |
                          <exists-CE> |
                          <forall-CE>
```

### 5.4.1 Pattern Conditional Element

**Pattern conditional elements** consist of a collection of field constraints, wildcards, and variables which are used to constrain the set of facts or instances which match the pattern CE. A pattern CE is satisfied by each and every pattern entity that satisfies its constraints. **Field constraints** are a set of constraints that are used to test a single field or slot of a pattern entity. A field constraint may consist of only a single literal constraint, however, it may also consist of

```
   =>
   (printout t "The system has a fault." crlf))

(defrule system-fault
   (error-status unknown)
   (valve broken)
   =>
   (printout t "The system has a fault." crlf))

(defrule system-fault
   (error-status unknown)
   (temp high)
   =>
   (printout t "The system has a fault." crlf))
```

### 5.4.4 And Conditional Element

CLIPS assumes that all rules have an implicit **and conditional element** surrounding the conditional elements on the LHS. This means that all conditional elements on the LHS must be satisfied before the rule can be activated. An explicit **and** conditional element is provided to allow the mixing of **and** CEs and **or** CEs. This allows other types of conditional elements to be grouped together within **or** and **not** CEs. The **and** CE is satisfied if *all* of the CEs inside of the explicit **and** CE are satisfied. If all other LHS conditions are true, the rule will be activated. Any number of conditional elements may be placed within an **and** CE.

Preview from Notesale.co.uk
Page 78 of 428

**Syntax**

```
<and-CE> ::= (and <conditional-element>+)
```

**Example**

```
(defrule system-flow
   (error-status confirmed)
   (or (and (temp high)
            (valve closed))
       (and (temp low)
            (valve open)))
   =>
   (printout t "The system is having a flow problem." crlf))
```

An **and** CE that has a **test** or **not** CE as its first CE has the pattern (initial-fact) or (initial-object) added as the first CE. Note that the LHS of any rule is enclosed within an implied **and** CE. For example, the following rule

```
(defrule nothing-to-schedule
  (not (schedule ?))
  =>
  (printout t "Nothing to schedule." crlf))
```

is converted to

```
(defrule nothing-to-schedule
```

```
(and (initial-fact)
     (not (schedule ?)))
=>
(printout t "Nothing to schedule." crlf))
```

### 5.4.5 Not Conditional Element

Sometimes the *lack* of information is meaningful; i.e., one wishes to fire a rule if a pattern entity or other CE does *not* exist. The **not conditional element** provides this capability. The **not** CE is satisfied only if the conditional element contained within it is not satisfied. As with other conditional elements, any number of additional CEs may be on the LHS of the rule and field constraints may be used within the negated pattern.

**Syntax**

```
<not-CE> ::= (not <conditional-element>)
```

Only one CE may be negated at a time. Multiple patterns may be negated by using multiple **not** CEs. Care must be taken when combining **not** CEs with **or** and **and** CEs; the results are not always obvious! The same holds true for variable bindings within a **not** CE. Previously bound variables may be used freely inside of a **not** CE. However, variables bound for the first time within a **not** CE can be used only in that pattern.

Preview from Notesale.co.uk
Page 79 of 428

**Examples**

```
(defrule high-flow-rate
   (temp high)
   (valve open)
   (not (error-status confirmed))
   =>
   (printout t "Recommend closing of valve due to high temp"
               crlf))

   (defrule check-valve
   (check-status ?valve)
   (not (valve-broken ?valve))
   =>
   (printout t "Device " ?valve " is OK" crlf))

(defrule double-pattern
   (data red)
   (not (data red ?x ?x))
   =>
   (printout t "No patterns with red green green!" crlf ))
```

A **not** CE that contains a single **test** CE is converted such that the **test** CE is contained within an **and** CE and is preceded by the (initial-fact) or (initial-object) pattern. For example, the following conditional element

```
(not (test (> ?time-1 ?time-2)))
```

### 5.4.8 Logical Conditional Element

The **logical conditional element** provides a **truth maintenance** capability for pattern entities (facts or instances) created by rules which use the **logical** CE. A pattern entity created on the RHS (or as a result of actions performed from the RHS) can be made logically dependent upon the pattern entities which matched the patterns enclosed with the **logical** CE on the LHS of the rule. The pattern entities matching the LHS **logical** patterns provide **logical support** to the facts and instance created by the RHS of the rule. A pattern entity can be logically supported by more than one group of pattern entities from the same or different rules. If any one supporting pattern entities is removed from a group of supporting pattern entities (and there are no other supporting groups), then the pattern entity is removed.

If a pattern entity is created without logical support (e.g., from a deffacts, definstaces, as a top-level command, or from a rule without any logical patterns), then the pattern entity has **unconditional support**. Unconditionally supporting a pattern entity removes all logical support (without causing the removal of the pattern entity). In addition, further logical support for an unconditionally supported pattern entity is ignored. Removing a rule that generated logical support for a pattern entity, removes the logical support generated by that rule (but does not cause the removal of the pattern entity if no logical support remains).

Preview from Notesale.co.uk
Page 84 of 428

**Syntax**

```
<logical-CE> ::= (logical <conditional-element>+)
```

The **logical** CE groups patterns together exactly as the explicit **and** CE does. It may be used in conjunction with the **and**, **or**, and **not** CE's. However, only the first N patterns of a rule can have the logical CE applied to them. For example, the following rule is legal

```
(defrule ok
   (logical (a))
   (logical (b))
   (c)
   =>
   (assert (d)))
```

whereas the following rules are illegal

```
(defrule not-ok-1
   (logical (a))
   (b)
   (logical (c))
   =>
   (assert (d)))

(defrule not-ok-2
   (a)
   (logical (b))
   (logical (c))
   =>
```

```
   (assert (d)))

(defrule not-ok-3
   (or (a)
       (logical (b)))
   (logical (c))
   =>
   (assert (d)))
```

**Example**

Given the following rules,

```
CLIPS> (clear)
CLIPS>
(defrule rule1
   (logical (a))
   (logical (b))
   (c)
   =>
   (assert (g) (h)))
CLIPS>
(defrule rule2
   (logical (d))
   (logical (e))
   (f)
   =>
   (assert (g) (h)))
CLIPS>
```

the following commands illustrate how logical dependencies work.

Preview from Notesale.co.uk
Page 85 of 428

```
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS. (watch rules)
CLIPS> (assert (a) (b) (c) (d) (e) (f))
==> f-0     (a)
==> f-1     (b)
==> f-2     (c)
==> Activation 0      rule1: f-0,f-1,f-2
==> f-3     (d)
==> f-4     (e)
==> f-5     (f)
==> Activation 0      rule2: f-3,f-3,f-5
<Fact-5>
CLIPS> (run)
FIRE    1 rule2: f-3,f-4,f-5 ; 1st rule adds logical support
==> f-6     (g)
==> f-7     (h)
FIRE    2 rule1: f-0,f-1,f-2 ; 2nd rule adds further support
CLIPS> (retract 1)
<== f-0     (a)           ; Removes 1st support for (g) and (h)
CLIPS> (assert (h))       ; (h) is unconditionally supported
FALSE
CLIPS> (retract 3)
<== f-3     (d)              ; Removes 2nd support for (g)
```

```
   (test (> 80 (startup-value)))
   (object (is-a MACHINE))
   =>)

(defrule example-4
   (machine ?x)
   (not (and (not (part ?x ?y))
             (inventoried ?x)))
   =>)
```

would be changed as follows.

```
(defrule example-2
   (initial-fact)
   (test (> 80 (startup-value)))
   =>)

(defrule example-3
   (object (is-a INITIAL-OBJECT) (name [initial-object]))
   (test (> 80 (startup-value)))
   (object (is-a MACHINE))
   =>)

(defrule example-4
   (machine ?x)
   (not (and (initial-fact)
             (not (part ?x ?y))
             (inventoried ?x)))
   =>)
```

Preview from Notesale.co.uk
Page 88 of 428

#### 5.4.9.3 Test CEs Following Not CEs

*Test* CEs that immediately follow a *not* CE are automatically moved by CLIPS behind the first pattern CE that precedes the *not* CE. For example, the following rule

```
(defrule example
   (a ?x)
   (not (b ?x))
   (test (> ?x 5))
   =>)
```

would be changed as follows.

```
(defrule example
   (a ?x)
   (test (> ?x 5))
   (not (b ?x))
   =>)
```

# Section 6 - Defglobal Construct

With the **defglobal** construct, global variables can be defined, set, and accessed within the CLIPS environment. Global variables can be accessed as part of the pattern-matching process, but changing them does not invoke the pattern-matching process. The **bind** function is used to set the value of global variables. Global variables are reset to their original value when the **reset** command is performed or when **bind** is called for the global with no values. This behavior can be changed using the **set-reset-globals** function. Global variables can be removed by using the **clear** command or the **undefglobal** command. If the globals item is being watched (see section 13.2), then an informational message will be displayed each time the value of a global variable is changed.

**Syntax**

```
(defglobal [<defmodule-name>] <global-assignment>*)

<global-assignment> ::= <global-variable> = <expression>

<global-variable>   ::= ?*<symbol>*
```

There may be multiple defglobal constructs and any number of global variables may be defined in each defglobal statement. The optional <defmodule-name> indicates the module in which the defglobals will be defined. If none is specified, the globals will be placed in the current module. If a variable was defined in a previous defglobal construct, its value will be replaced by the value found in the new defglobal construct. If an error is encountered when defining a defglobal construct, any global variable definitions that occurred before the error was encountered will still remain in effect.

Preview from Notesale.co.uk
Page 93 of 428

Commands that operate on defglobals such as ppdefglobal and undefglobal expect the symbolic name of the global without the astericks (e.g. use the symbol *max* when you want to refer to the global variable ?*max*).

Global variables may be used anyplace that a local variable could be used (with two exceptions). Global variables may not be used as a parameter variable for a deffunction, defmethod, or message-handler. Global variables may not be used in the same way that a local variable is used on the LHS of a rule to bind a value. Therefore, the following rule is illegal

```
(defrule example
   (fact ?*x*)
   =>)
```

The following rule, however, is legal.

```
(defrule example
   (fact ?y&:(> ?y ?*x*))
   =>)
```

```
(defrule collect-factoids
   (collect-factoids)
   =>
   (bind ?data (create$))
   (do-for-all-facts ((?f factoid)) TRUE
      (bind ?data (create$ ?data ?f:implied)))
   (assert (collection ?data)))
```

With this approach, the *collection* fact is available for pattern-matching with the added benefit that there are no intermediate results generated in creating the fact. Typically if other rules are waiting for the finished result of the collection, they would need to have lower salience so that they aren't fired for the intermediate results:

```
(defrule print-factoids
   (declare (salience -10))
   (collection $?data)
   =>
   (printout t "The collected data is " ?data crlf))
```

If the *factoid* facts are collected by a single rule firing, then the salience declaration is unnecessary.

Preview from Notesale.co.uk
Page 96 of 428

**Appropriate Uses**

The primary use of global variables (in conjunction with rules) is in making a program easier to maintain. It is a rare situation where a global variable is required in order to solve a problem. One appropriate use of global variables is defining salience values shared among multiple rules:

```
(defglobal ?*high-priority* = 100)

(defrule rule-1
   (declare (salience ?*high-priority*))
   =>)

(defrule rule-2
   (declare (salience ?*high-priority*))
   =>)
```

Another use is defining constants used on the LHS or RHS of a rule:

```
(defglobal ?*week-days* =
   (create$ monday tuesday wednesday thursday friday saturday sunday))

(defrule invalid-day
   (day ?day&:(not (member$ ?day ?*week-days*)))
   =>
   (printout t ?day " is invalid" crlf))

(defrule valid-day
   (day ?day&:(member$ ?day ?*week-days*))
   =>
   (printout t ?day " is valid" crlf))
```

A third use is passing information to a rule when it is desirable *not* to trigger pattern-matching. In the following rule, a global variable is used to determine whether additional debugging information is printed:

```
(defglobal ?*debug-print* = nil)

(defrule rule-debug
   ?f <- (info ?info)
   =>
   (retract ?f)
   (printout ?*debug-print* "Retracting info " ?info crlf))
```

If ?*debug-print* is set to nil, then the printout statement will not display any information. If the ?*debug-print* is set to t, then debugging information will be sent to the screen. Because ?*debug-print* is a global, it can be changed interactively without causing rules to be reactivated. This is useful when stepping through a program because it allows the level of information displayed to be changed without effecting the normal flow of the program.

It's possible, but a little more verbose, to achieve this same functionality using instances rather than global variables:

Preview from Notesale.co.uk
Page 97 of 428

```
(defclass DEBUG-INFO
   (is-a USER)
   (slot debug-print))

(definstances debug
   ([debug-info] of DEBUG-INFO (debug-print nil)))

(defrule rule-debug
   ?f <- (info ?info)
   =>
   (retract ?f)
   (printout (send [debug-info] get-debug-print) "Retracting info " ?info crlf))
```

Unlike fact slots, changes to a slot of an instance won't trigger pattern matching in a rule unless the slot is specified on the LHS of that rule, thus you have explicit control over whether an instance slot triggers pattern-matching. The following rule won't be retriggered if a change is made to the *debug-print* slot:

```
(defrule rule-debug
   ?f <- (info ?info)
   (object (is-a DEBUG-INFO) (name ?name))
   =>
   (retract ?f)
   (printout (send ?name get-debug-print) "Retracting info " ?info crlf))
```

This is a generally applicable technique and can be used in many situations to prevent rules from inadvertently looping when slot values are changed.

```
<slot>  ::= (slot <name> <facet>*) |
            (single-slot <name> <facet>*) |
            (multislot <name> <facet>*)

<facet> ::=  <default-facet> | <storage-facet> |
             <access-facet> | <propagation-facet> |
             <source-facet> | <pattern-match-facet> |
             <visibility-facet> | <create-accessor-facet>
             <override-message-facet> | <constraint-attributes>

<default-facet> ::=
           (default ?DERIVE | ?NONE | <expression>*) |
           (default-dynamic <expression>*)

<storage-facet> ::= (storage local | shared)

<access-facet>
       ::= (access read-write | read-only | initialize-only)

<propagation-facet> ::= (propagation inherit | no-inherit)

<source-facet> ::= (source exclusive | composite)

<pattern-match-facet>
       ::= (pattern-match reactive | non-reactive)

<visibility-facet> ::= (visibility private | public)

<create-accessor-facet>
     ::= (create-accessor ?NONE | read | write | read-write)

<override-message-facet>
     ::= (override-message ?DEFAULT | <message-name>)

<handler-documentation>
       ::= (message-handler <name> [<handler-type>])

<handler-type> ::= primary | around | before | after
```

Preview from Notesale.co.uk
Page 115 of 428

Redefining an existing class deletes the current subclasses and all associated message-handlers. An error will occur if instances of the class or any of its subclasses exist.

### 9.3.1 Multiple Inheritance

If one class inherits from another class, the first class is a **subclass** of the second class, and the second class is a **superclass** of the first class. Every user-defined class must have at least one direct superclass, i.e. at least one class must appear in the *is-a* portion of the defclass. Multiple inheritance occurs when a class has more than one direct superclass. COOL examines the direct superclass list for a new class to establish a linear ordering called the **class precedence list**. The new class inherits slots and message-handlers from each of the classes in the class precedence list. The word precedence implies that slots and message-handlers of a class in the list override

Class D directly inherits information from the classes B and A. The class precedence list for D is: D B A USER OBJECT.

**Example 5**

```
(defclass E (is-a A C))
```

By rule #2, A must precede C. However, C is a subclass of A and cannot succeed A in a precedence list without violating rule #1. Thus, this is an error.

**Example 6**

```
(defclass E (is-a C A))
```

Specifying that E inherits from A is extraneous, since C inherits from A. However, this definition does not violate any rules and is acceptable. The class precedence list for E is: E C A B USER OBJECT.

**Example 7**

```
(defclass F (is-a C B))
```

Specifying that F inherits from B is extraneous, since C inherits from B. The class precedence list for F is: F C A B USER OBJECT. The superclass list says B must follow C in F's class precedence list but *not* that B must *immediately* follow C.

Preview from Notesale.co.uk
Page 117 of 428

**Example 8**

```
(defclass G (is-a C D))
```

This is an error, for it violates rule #2. The class precedence of C says that A should precede B, but the class precedence list of D says the opposite.

**Example 9**

```
(defclass H (is-a A))
(defclass I (is-a B))
(defclass J (is-a H I A B))
```

The respective class precedence lists of H and I are: H A USER OBJECT and I B USER OBJECT. If J did not have A and B as direct superclasses, J could have one of three possible class precedence lists: J H A I B USER OBJECT, J H I A B USER OBJECT or J H I B A USER OBJECT. COOL would normally pick the first list since it preserves the family trees (H A and I B) to the greatest extent possible. However, since J inherits directly from A and B, rule #2 dictates that the class precedence list must be J H I A B USER OBJECT.

**Usage Note**

For most practical applications of multiple inheritance, the order in which the superclasses are specified should not matter. If you create a class using multiple inheritance and the order of the

and read. The **read-only** facet says the slot can only be read; the only way to set this slot is with default facets in the class definition. The **initialize-only** facet is like **read-only** except that the slot can also be set by slot overrides in a **make-instance** call (see section 9.6.1) and **init** message-handlers (see section 9.4). These privileges apply to indirect access via messages as well as direct access within message-handler bodies (see section 9.4). Note: a **read-only** slot that has a static default value will implicitly have the **shared** storage facet.

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (role concrete)
   (slot foo (create-accessor write)
             (access read-write))
   (slot bar (access read-only)
             (default abc))
   (slot woz (create-accessor write)
             (access initialize-only)))
CLIPS>
(defmessage-handler A put-bar (?value)
   (dynamic-put (sym-cat bar) ?value))
CLIPS> (make-instance a of A (bar 34))
[MSGFUN3] bar slot in [a] of A: write access denied.
[PRCCODE4] Execution halted during the actions of message-handler put-bar primary
in class A
FALSE
CLIPS> (make-instance a of A (foo 34) (woz 65))
[a]
CLIPS> (send [a] put-bar 1)
[MSGFUN3] bar slot in [a] of A: write access denied.
[PRCCODE4] Execution halted during the actions of message-handler put-bar primary
in class A
FALSE
CLIPS> (send [a] put-woz 1)
[MSGFUN3] woz slot in [a] of A: write access denied.
[PRCCODE4] Execution halted during the actions of message-handler put-bar primary
in class A
FALSE
CLIPS> (send [a] print)
[a] of A
(foo 34)
(bar abc)
(woz 65)
CLIPS>
```

Preview from Notesale.co.uk
Page 122 of 428

### 9.3.3.5 Inheritance Propagation Facet

An **inherit** facet says that a slot in a class can be given to instances of other classes that inherit from the first class. This is the default. The **no-inherit** facet says that only direct instances of this class will get the slot.

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (role concrete)
   (slot foo (propagation inherit))
   (slot bar (propagation no-inherit)))
CLIPS> (defclass B (is-a A))
CLIPS> (make-instance a of A)
[a]
CLIPS> (make-instance b of B)
[b]
CLIPS> (send [a] print)
[a] of A
(foo nil)
(bar nil)
CLIPS> (send [b] print)
[b] of B
(foo nil)
CLIPS>
```

### 9.3.3.6 Source Facet

When obtaining slots from the class precedence list during instance creation, the default behavior is to take the facets from the most specific class which gives the slot and give default values to any unspecified facets. This is the behavior specified by the **exclusive** facet. The **composite** facet causes facets which are not explicitly specified by the most specific class to be taken from the next most specific class. Thus, in an overlay fashion, the facets of an instance's slot can be specified by more than one class. Note that even though facets may be taken from superclasses, the slot is still considered to reside in the new class for purposes of visibility (see section 9.3.3.8). One good example of a use of this feature is to pick up a slot definition and change only its default value for a new derived class.

Preview from Notesale.co.uk
Page 123 of 428

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (multislot foo (access read-only)
                 (default a b c)))
CLIPS>
(defclass B (is-a A)
   (slot foo (source composite) ; multiple and read-only
                                ; from class A
                 (default d e f)))
CLIPS> (describe-class B)
==============================================================
**************************************************************
Concrete: direct instances of this class can be created.
Reactive: direct instances of this class can match defrule patterns.

Direct Superclasses: A
Inheritance Precedence: B A USER OBJECT
```

**Example**

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS> (make-instance a of A)
[a]
CLIPS>
(defmessage-handler A print-args (?a ?b $?c)
   (printout t (instance-name ?self) " " ?a " " ?b
          " and " (length$ ?c) " extras: " ?c crlf))
CLIPS> (send [a] print-args 1 2)
[a] 1 2 and 0 extras: ()
CLIPS> (send [a] print-args a b c d)
[a] a b and 2 extras: (c d)
CLIPS>
```

### 9.4.2 Message-handler Actions

The body of a message-handler is a sequence of expressions that are executed in order when the handler is called. The return value of the message-handler is the result of the evaluation of the last expression in the body.

Handler actions may *directly* manipulate slots of the active instance. Normally, slots can only be manipulated by sending the object slot-accessor messages (see sections 9.3.3.9 and 9.4.3). However, handlers are considered part of the encapsulation (see section 2.6.2) of an object, and thus can directly view and change the slots of the object. There are several functions which operate implicitly on the active instance (without the use of messages) and can only be called from within a message-handler. These functions are discussed in section 12.16.

Preview from Notesale.co.uk
Page 132 of 428

A shorthand notation is provided for accessing slots of the active instance from within a message-handler.

**Syntax**

```
?self:<slot-name>
```

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (role concrete)
  (slot foo (default 1))
  (slot bar (default 2)))
CLIPS>
(defmessage-handler A print-all-slots ()
  (printout t ?self:foo " " ?self:bar crlf))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print-all-slots)
1 2
CLIPS>
```

```
(defclass B (is-a A)
  (role concrete)
  (slot foo (visibility public)))
CLIPS> (make-instance b of B)
[b]
CLIPS> (send [b] get-foo)
nil
CLIPS>
```

### 9.4.3 Daemons

Daemons are pieces of code which execute implicitly whenever some basic action is taken upon an instance, such as initialization, deletion, or reading and writing of slots. All these basic actions are implemented with primary handlers attached to the class of the instance. Daemons may be easily implemented by defining other types of message-handlers, such as before or after, which will recognize the same messages. These pieces of code will then be executed whenever the basic actions are performed on the instance.

**Example**

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS>
(defmessage-handler A init before ()
   (printout t "Initializing a new instance of class A..."
             crlf))
CLIPS> (make-instance a of A)
Initializing a new instance of class A...
[a]
CLIPS>
```

Preview from Notesale.co.uk
Page 134 of 428

### 9.4.4 Predefined System Message-handlers

CLIPS defines eight primary message-handlers that are attached to the class USER. These handlers cannot be deleted or modified.

#### 9.4.4.1 Instance Initialization

**Syntax**

```
(defmessage-handler USER init primary ())
```

This handler is responsible for initializing instances with class default values after creation. The **make-instance** and **initialize-instance** functions send the **init** message to an instance (see sections 9.6.1 and 9.6.2); the user should never send this message directly. This handler is implemented using the **init-slots** function (see section 12.13). User-defined **init** handlers should not prevent the system message-handler from responding to an **init** message (see section 9.5.3).

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass CAR (is-a USER)
  (role concrete)
  (slot price (default 75000))
  (slot model (default Corniche)))
CLIPS> (watch messages)
CLIPS> (watch message-handlers)
CLIPS> (make-instance Rolls-Royce of CAR)
MSG >> create ED:1 (<Instance-Rolls-Royce>)
HND >> create primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
HND << create primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
MSG << create ED:1 (<Instance-Rolls-Royce>)
MSG >> init ED:1 (<Instance-Rolls-Royce>)
HND >> init primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
HND << init primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
MSG << init ED:1 (<Instance-Rolls-Royce>)
[Rolls-Royce]
CLIPS>
```

Preview from Notesale.co.uk
Page 135 of 428

### 9.4.4.2 Instance Deletion

**Syntax**

```
(defmessage-handler USER delete primary ())
```

This handler is responsible for deleting an instance from the system. The user must directly send a **delete** message to an instance. User-defined **delete** message-handlers should not prevent the system message-handler from responding to a **delete** message (see section 9.5.3). The handler returns the symbol TRUE if the instance was successfully deleted, otherwise it returns the symbol FALSE.

**Example**

```
CLIPS> (send [Rolls-Royce] delete)
MSG >> delete ED:1 (<Instance-Rolls-Royce>)
HND >> delete primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
HND << delete primary in class USER
       ED:1 (<Stale Instance-Rolls-Royce>)
MSG << delete ED:1 (<Stale Instance-Rolls-Royce>)
TRUE
CLIPS>
```

### 9.4.4.3 Instance Display

**Syntax**

```
(defmessage-handler USER print primary ())
```

This handler prints out slots and their values for an instance.

**Example**

```
CLIPS> (make-instance Rolls-Royce of CAR)
MSG >> create ED:1 (<Instance-Rolls-Royce>)
HND >> create primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
HND << create primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
MSG << create ED:1 (<Instance-Rolls-Royce>)
MSG >> init ED:1 (<Instance-Rolls-Royce>)
HND >> init primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
HND << init primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
MSG << init ED:1 (<Instance-Rolls-Royce>)
[Rolls-Royce]
CLIPS> (send [Rolls-Royce] print)
MSG >> print ED:1 (<Instance-Rolls-Royce>)
HND >> print primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
[Rolls-Royce] of CAR
(price 75000)
(model Cornniche)
HND << print primary in class USER
       ED:1 (<Instance-Rolls-Royce>)
MSG << print ED:1 (<Instance-Rolls-Royce>)
CLIPS> (unwatch messages)
CLIPS. (unwatch message-handlers)
CLIPS>
```

Preview from Notesale.co.uk
Page 136 of 428

### 9.4.4.4 Directly Modifying an Instance

**Syntax**

```
(defmessage-handler USER direct-modify primary
  (?slot-override-expressions))
```

This handler modifies the slots of an instance directly rather than using put- override messages to place the slot values. The slot-override expressions are passed as an EXTERNAL_ADDRESS data object to the direct-modify handler. This message is used by the functions **modify-instance** and **active-modify-instance**.

**Example**

The following around message-handler could be used to insure that all modify message slot-overrides are handled using put- messages.

```
(defmessage-handler USER direct-modify around
   (?overrides)
   (send ?self message-modify ?overrides))
```

### 9.4.4.5 Modifying an Instance using Messages

**Syntax**

```
(defmessage-handler USER message-modify primary
  (?slot-override-expressions)
```

This handler modifies the slots of an instance using put- messages for each slot update. The slot-override expressions are passed as an EXTERNAL_ADDRESS data object to the message-modify handler. This message is used by the functions **message-modify-instance** and **active-message-modify-instance**.

### 9.4.4.6 Directly Duplicating an Instance

Preview from Notesale.co.uk
Page 137 of 428

**Syntax**

```
(defmessage-handler USER direct-duplicate primary
  (?new-instance-name ?slot-override-expressions))
```

This handler duplicates an instance without using put- messages to assign the slot-overrides. Slot values from the original instance and slot overrides are directly copied. If the name of the new instance created matches a currently existing instance-name, then the currently existing instance is deleted without use of a message. The slot-override expressions are passed as an EXTERNAL_ADDRESS data object to the direct-duplicate handler. This message is used by the functions **duplicate-instance** and **active-duplicate-instance**.

**Example**

The following around message-handler could be used to insure that all duplicate message slot-overrides are handled using put- messages.

```
(defmessage-handler USER direct-duplicate around
   (?new-name ?overrides)
   (send ?self message-duplicate ?new-name ?overrides))
```

There must be at least one applicable primary handler for a message, or a message execution error will be generated (see section 9.5.4).

### 9.5.3 Shadowed Message-handlers

When one handler must be called by another handler in order to be executed, the first handler is said to be **shadowed** by the second. An around handler shadows all handlers except more specific around handlers. A primary handler shadows all more general primary handlers.

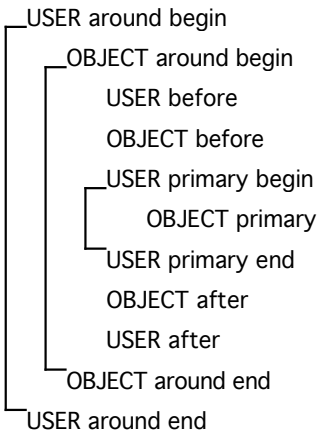
Messages should be implemented using the declarative technique, if possible. Only the handler roles will dictate which handlers get executed; only before and after handlers and the most specific primary handler are used. This allows each handler for a message to be completely independent of the other message-handlers. However, if around handlers or shadowed primary handlers are necessary, then the handlers must explicitly take part in the message dispatch by calling other handlers they are shadowing. This is called the imperative technique. The functions **call-next-handler** and **override-next-handler** (see section 12.16.2) allow a handler to execute the handler it is shadowing. A handler can call the same shadowed handler multiple times.

Preview from Notesale.co.uk
Page 140 of 428

**Example**

```
(defmessage-handler USER my-message around ()
     (call-next-handler))
(defmessage-handler USER my-message before ())
(defmessage-handler USER my-message ()
     (call-next-handler))
(defmessage-handler USER my-message after ())
(defmessage-handler OBJECT my-message around ()
     (call-next-handler))
(defmessage-handler OBJECT my-message before ())
(defmessage-handler OBJECT my-message ())
(defmessage-handler OBJECT my-message after ())
```

For a message sent to an instance of a class which inherits from USER, the diagram to the right illustrates the order of execution for the handlers attached to the classes USER and OBJECT. The brackets indicate where a particular handler begins and ends execution. Handlers enclosed within a bracket are shadowed.




### 9.5.4 Message Execution Errors

If an error occurs at any time during the execution of a message-handler, any currently executing handlers will be aborted, any handlers which have not yet started execution will be ignored, and the **send** function will return the symbol FALSE.

```
CLIPS>
(defclass A (is-a USER) (role concrete)
   (slot x (create-accessor write) (default 1)))
CLIPS>
(definstances A-OBJECTS
   (a1 of A)
   (of A (x 65)))
CLIPS> (watch instances)
CLIPS> (reset)
==> instance [initial-object] of INITIAL-OBJECT
==> instance [a1] of A
==> instance [gen1] of A
CLIPS> (reset)
<== instance [initial-object] of INITIAL-OBJECT
<== instance [a1] of A
<== instance [gen1] of A
==> instance [initial-object] of INITIAL-OBJECT
==> instance [a1] of A
==> instance [gen2] of A
CLIPS> (unwatch instances)
CLIPS>
```

Upon startup and after a **clear** command, CLIPS automatically constructs the following definstances.

```
(definstances initial-object
   (initial-object of INITIAL-OBJECT))
```

The class INITIAL-OBJECT is a predefined system class that is a direct subclass of USER.

```
(defclass INITIAL-OBJECT
   (is-a USER)
   (role concrete)
   (pattern-match reactive))
```

Preview from Notesale.co.uk
Page 144 of 428

The initial-object definstances and the INITIAL-OBJECT class are only defined if both the object system and defrules are enabled (see section 2 of the *Advanced Programming Guide*). The INITIAL-OBJECT class cannot be deleted, but the *initial-object* definstances can. See section 5.4.9 for details on default patterns which pattern-match against the *initial-object* instance.

**Important Note**

Although you can delete the *initial-object* definstances, in practice you never should since many conditional elements rely on the existence of the *initial-object* instance for correct operation. Similarly, the *initial-object* instance created by the *initial-object* definstances when a **reset** command is issued, should never be deleted by a program.

### 9.6.2 Reinitializing Existing Instances

The **initialize-instance** function provides the ability to reinitialize an existing instance with class defaults and new slot-overrides. The return value of **initialize-instance** is the name of the

```
HND >> direct-modify primary in class USER.
       ED:1 (<Instance-a> <Pointer-0019CD5A>)
::= local slot foo in instance a <- 0
HND << direct-modify primary in class USER.
       ED:1 (<Instance-a> <Pointer-0019CD5A>)
MSG << direct-modify ED:1 (<Instance-a> <Pointer-0019CD5A>)
TRUE
CLIPS> (unwatch all)
CLIPS>
```

#### 9.6.7.2 Directly Modifying an Instance with Immediate Pattern-Matching

The **active-modify-instance** function uses the **direct-modify** message to change the values of the instance. Object pattern-matching occurs as slot modifications are being performed.

**Syntax**

```
(active-modify-instance <instance> <slot-override>*)
```

#### 9.6.7.3 Modifying an Instance using Messages with Delayed Pattern-Matching

The **message-modify-instance** function uses the **message-modify** message to change the values of the instance. Object pattern-matching is delayed until all of the slot modifications have been performed.

Preview from Notesale.co.uk
Page 149 of 428

**Syntax**

```
(message-modify-instance <instance> <slot-override>*)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (role concrete)
  (slot foo)
  (slot bar (create-accessor write)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (watch all)
CLIPS> (message-modify-instance a (bar 4))
MSG >> message-modify ED:1 (<Instance-a> <Pointer-009F04A0>)
HND >> message-modify primary in class USER
       ED:1 (<Instance-a> <Pointer-009F04A0>)
MSG >> put-bar ED:2 (<Instance-a> 4)
HND >> put-bar primary in class A
       ED:2 (<Instance-a> 4)
::= local slot bar in instance a <- 4
HND << put-bar primary in class A
       ED:2 (<Instance-a> 4)
MSG << put-bar ED:2 (<Instance-a> 4)
HND << message-modify primary in class USER
       ED:1 (<Instance-a> <Pointer-009F04A0>)
MSG << message-modify ED:1 (<Instance-a> <Pointer-009F04A0>)
```

```
<== instance [b] of A
HND << delete primary in class USER
       ED:2 (<Stale Instance-b>)
MSG << delete ED:2 (<Stale Instance-b>)
==> instance [b] of A
MSG >> create ED:2 (<Instance-b>)
HND >> create primary in class USER
       ED:2 (<Instance-b>)
HND << create primary in class USER
       ED:2 (<Instance-b>)
MSG << create ED:2 (<Instance-b>)
MSG >> put-bar ED:2 (<Instance-b> 6)
HND >> put-bar primary in class A
       ED:2 (<Instance-b> 6)
::= local slot bar in instance b <- 6
HND << put-bar primary in class A
       ED:2 (<Instance-b> 6)
MSG << put-bar ED:2 (<Instance-b> 6)
MSG >> put-foo ED:2 (<Instance-b> 0)
HND >> put-foo primary in class A
       ED:2 (<Instance-b> 0)
::= local slot foo in instance b <- 0
HND << put-foo primary in class A
       ED:2 (<Instance-b> 0)
MSG << put-foo ED:2 (<Instance-b> 0)
MSG >> init ED:2 (<Instance-b>)
HND >> init primary in class USER
       ED:2 (<Instance-b>)
HND << init primary in class USER
       ED:2 (<Instance-b>)
MSG << init ED:2 (<Instance-b>)
HND << message-duplicate primary in class USER
       ED:1 (<Instance-a> [b] <Pointer-009F04A0>)
MSG << message-duplicate ED:1 (<Instance-a> [b] <Pointer-009F04A0>)
[b]
CLIPS> (unwatch all)
CLIPS>
```

Preview from Notesale.co.uk
Page 152 of 428

### 9.6.8.4 Duplicating an Instance using Messages with Immediate Pattern-Matching

The **active-message-duplicate-instance** function uses the **message-duplicate** message to change the values of the instance. Object pattern-matching occurs as slot modifications are being performed.

**Syntax**

```
(active-message-duplicate-instance <instance>
                                   [to <instance-name>]
                                   <slot-override>*)
```

**Example**

For the instance-set template given in section 9.7.1, thirty instance-sets would be generated in the following order:

```
1.  [Boy-1] [Girl-1]
2.  [Boy-1] [Girl-2]
3.  [Boy-1] [Woman-1]
4.  [Boy-1] [Woman-2]
5.  [Boy-1] [Woman-3]
6.  [Boy-2] [Girl-1]
7.  [Boy-2] [Girl-2]
8.  [Boy-2] [Woman-1]
9.  [Boy-2] [Woman-2]
10. [Boy-2] [Woman-3]
11. [Boy-3] [Girl-1]
12. [Boy-3] [Girl-2]
13  [Boy-3] [Woman-1]
14. [Boy-3] [Woman-2]
15. [Boy-3] [Woman-3]
16. [Boy-4] [Girl-1]
17. [Boy-4] [Girl-2]
18. [Boy-4] [Woman-1]
19. [Boy-4] [Woman-2]
20. [Boy-4] [Woman-3]
21. [Man-1] [Girl-1]
22. [Man-1] [Girl-2]
23. [Man-1] [Woman-1]
24. [Man-1] [Woman-2]
25. [Man-1] [Woman-3]
26. [Man-2] [Girl-1]
27. [Man-2] [Girl-2]
28. [Man-2] [Woman-1]
29. [Man-2] [Woman-2]
30. [Man-2] [Woman-3]
```

**Example**

Consider the following instance-set template:

```
((?f1 FEMALE) (?f2 FEMALE))
```

Twenty-five instance-sets would be generated in the following order:

```
1. [Girl-1] [Girl-1]
2. [Girl-1] [Girl-2]
3. [Girl-1] [Woman-1]
4. [Girl-1] [Woman-2]
5. [Girl-1] [Woman-3]
6. [Girl-2] [Girl-1]
7. [Girl-2] [Girl-2]
8. [Girl-2] [Woman-1]
9. [Girl-2] [Woman-2]
10.[Girl-2] [Woman-3]
11.[Woman-1] [Girl-1]
12.[Woman-1] [Girl-2]
13.[Woman-1] [Woman-1]
14.[Woman-1] [Woman-2]
15.[Woman-1] [Woman-3]
16.[Woman-2] [Girl-1]
17.[Woman-2] [Girl-2]
18.[Woman-2] [Woman-1]
19.[Woman-2] [Woman-2]
20.[Woman-2] [Woman-3]
21.[Woman-3] [Girl-1]
22.[Woman-3] [Girl-2]
23.[Woman-3] [Woman-1]
24.[Woman-3] [Woman-2]
25.[Woman-3] [Woman-3]
```

Preview from Notesale.co.uk
Page 156 of 428

The instances of class GIRL are examined before the instances of class WOMAN because GIRL was defined before WOMAN.

### 9.7.3 Query Definition

A **query** is a user-defined boolean expression applied to an instance-set to determine if the instance-set meets further user-defined restrictions. If the evaluation of this expression for an instance-set is anything but the symbol FALSE, the instance-set is said to satisfy the query.

**Syntax**

```
<query> ::= <boolean-expression>
```

**Example**

Continuing the previous example, one query might be that the two instances in an ordered pair have the same age.

```
(= (send ?man-or-boy get-age) (send ?woman-or-girl get-age))
```

Within a query, slots of instance-set members can be directly read with a shorthand notation similar to that used in message-handlers (see section 9.4.2). If message-passing is not explicitly required for reading a slot (i.e. there are no accessor daemons for reads), then this second method of slot access should be used, for it gives a significant performance benefit.

**Syntax**

```
<instance-set-member-variable>:<slot-name>
```

**Example**

The previous example could be rewritten as:

```
(= ?man-or-boy:age ?woman-or-girl:age)
```

Since only instance-sets which satisfy a query are of interest and the query is evaluated for all possible instance-sets, the query should not have any side-effects.

Preview from Notesale.co.uk
Page 157 of 428

### 9.7.4 Distributed Action Definition

A **distributed action** is a user-defined expression evaluated for each instance-set which satisfies a query. Unlike queries, distributed actions must use messages to read slots of instance-set members. If more than one action is required, use the **progn** function (see section 12.6.5) to group them.

**Action Syntax**

```
<action> ::= <expression>
```

**Example**

Continuing the previous example, one distributed action might be to simply print out the ordered pair to the screen.

```
(printout t "(" ?man-or-boy "," ?woman-or-girl ")" crlf)
```

### 9.7.5 Scope in Instance-set Query Functions

An instance-set query function can be called from anywhere that a regular function can be called. If a variable from an outer scope is not masked by an instance-set member variable, then that variable may be referenced within the query and action. In addition, rebinding variables within an instance-set function action is allowed. However, attempts to rebind instance-set member variables will generate errors. Binding variables is not allowed within a query. Instance-set query functions can be nested.

**Example**

```
CLIPS>
(deffunction count-instances (?class)
  (bind ?count 0)
  (do-for-all-instances ((?ins ?class)) TRUE
    (bind ?count (+ ?count 1)))
 ?count)
CLIPS>
(deffunction count-instances-2 (?class)
  (length (find-all-instances ((?ins ?class)) TRUE)))
CLIPS> (count-instances WOMAN)
3
CLIPS> (count-instances-2 BOY)
4
CLIPS>
```

Instance-set member variables are only in scope within the instance-set query function. Attempting to use instance-set member variables in an outer scope will generate an error.

Preview from Notesale.co.uk
Page 158 of 428

**Example**

```
CLIPS>
(deffunction last-instance (?class)
   (any-instancep ((?ins ?class)) TRUE)
   ?ins)

[PRCCODE3] Undefined variable ins referenced in deffunction.

ERROR:
(deffunction last-instance
   (?class)
   (any-instancep ((?ins ?class))
      TRUE)
   ?ins
   )
CLIPS>
```

### 9.7.6 Errors during Instance-set Query Functions

If an error occurs during an instance-set query function, the function will be immediately terminated and the return value will be the symbol FALSE.

### 9.7.7 Halting and Returning Values from Query Functions

The functions **break** and **return** are now valid inside the action of the instance-set query functions **do-for-instance**, **do-for-all-instances** and **delayed-do-for-all-instances**. The **return** function is only valid if it is applicable in the outer scope, whereas the **break** function actually halts the query.

### 9.7.8 Instance-set Query Functions

The instance query system in COOL provides six functions. For a given set of instances, all six query functions will iterate over these instances in the same order (see section 9.7.2). However, if a particular instance is deleted and recreated, the iteration order will change.

#### 9.7.8.1 Testing if Any Instance-set Satisfies a Query

This function applies a query to each instance-set which matches the template. If an instance-set satisfies the query, then the function is immediately terminated, and the return value is the symbol TRUE. Otherwise, the return value is the symbol FALSE.

**Syntax**

```
(any-instancep <instance-set-template> <query>)
```

**Example**

Are there any men over age 30?

```
CLIPS> (any-instancep ((?man MAN)) (> ?man:age 30))
TRUE
CLIPS>
```

Preview from Notesale.co.uk
Page 159 of 428

#### 9.7.8.2 Determining the First Instance-set Satisfying a Query

This function applies a query to each instance-set which matches the template. If an instance-set satisfies the query, then the function is immediately terminated, and the instance-set is returned in a multifield value. Otherwise, the return value is a zero-length multifield value. Each field of the multifield value is an instance-name representing an instance-set member.

**Syntax**

```
(find-instance <instance-set-template> <query>)
```

**Example**

Find the first pair of a man and a woman who have the same age.

```
CLIPS>
(find-instance ((?m MAN) (?w WOMAN)) (= ?m:age ?w:age))
([Man-1] [Woman-1])
```

```
CLIPS>
```

### 9.7.8.3 Determining All Instance-sets Satisfying a Query

This function applies a query to each instance-set which matches the template. Each instance-set which satisfies the query is stored in a multifield value. This multifield value is returned when the query has been applied to all possible instance-sets. If there are n instances in each instance-set, and m instance-sets satisfied the query, then the length of the returned multifield value will be n * m. The first n fields correspond to the first instance-set, and so on. Each field of the multifield value is an instance-name representing an instance-set member. The multifield value can consume a large amount of memory due to permutational explosion, so this function should be used judiciously.

**Syntax**

```
(find-all-instances <instance-set-template> <query>)
```

**Example**

Find all pairs of a man and a woman who have the same age.

```
CLIPS>
(find-all-instances ((?m MAN) (?w WOMAN)) (= ?m:age ?w:age))
([Man-1] [Woman-1] [Man-2] [Woman-2])
CLIPS>
```

Preview from Notesale.co.uk
Page 160 of 428

### 9.7.8.4 Executing an Action for the First Instance-set Satisfying a Query

This function applies a query to each instance-set which matches the template. If an instance-set satisfies the query, the specified action is executed, and the function is immediately terminated. The return value is the evaluation of the action. If no instance-set satisfied the query, then the return value is the symbol FALSE.

**Syntax**

```
(do-for-instance <instance-set-template> <query> <action>*)
```

**Example**

Print out the first triplet of different people that have the same age. The calls to **neq** in the query eliminate the permutations where two or more members of the instance-set are identical.

```
CLIPS>
(do-for-instance ((?p1 PERSON) (?p2 PERSON) (?p3 PERSON))
  (and (= ?p1:age ?p2:age ?p3:age)
       (neq ?p1 ?p2)
       (neq ?p1 ?p3)
       (neq ?p2 ?p3))
  (printout t ?p1 " " ?p2 " " ?p3 crlf))
[Girl-2] [Boy-2] [Boy-3]
CLIPS>
```

**Example**

```
(defmodule FOO
   (import BAR ?ALL)
   (import YAK deftemplate ?ALL)
   (import GOZ defglobal x y z)
   (export defgeneric +)
   (export defclass ?ALL))
```

## 10.2 SPECIFYING A CONSTRUCT'S MODULE

The module in which a construct is placed can be specified when the construct is defined. The deffacts, deftemplate, defrule, deffunction, defgeneric, defclass, and definstances constructs all specify the module for the construct by including it as part of the name. The module of a defglobal construct is indicated by specifying the module name after the defglobal keyword. The module of a defmessage-handler is specified as part of the class specifier. The module of a defmethod is specified as part of the generic function specifier. For example, the following constructs would be placed in the DETECTION module.

```
(defrule DETECTION::Find-Fault
   (sensor (name ?name) (value bad))
   =>
   (assert (fault (name ?name))))

(defglobal DETECTION ?*count* = 0)

(defmessage-handler DETECTION::COMPONENT get-charge ()
   (* ?self:flux ?self:flow))

(defmethod DETECTION::+ ((?x STRING) (?y STRING))
   (str-cat ?x ?y))
```

Preview from Notesale.co.uk
Page 164 of 428

**Example**

```
CLIPS> (clear)
CLIPS> (defmodule A)
CLIPS> (defmodule B)
CLIPS> (defrule foo =>)
CLIPS> (defrule A::bar =>)
CLIPS> (list-defrules)
bar
For a total of 1 defrule.
CLIPS> (set-current-module B)
A
CLIPS> (list-defrules)
foo
For a total of 1 defrule.
CLIPS>
```

knowledge base to be partitioned such that rules and other constructs can only "see" those facts and instances which are of interest to them. Note that the *initial-fact* deftemplate and the INITIAL-OBJECT defclass must explicitly be imported from the MAIN module. Rules which have the *initial-fact* or *initial-object* pattern added to their LHS (such as a rule thats first CE is a *not* CE) will not be activated unless the corresponding construct for the pattern is imported.

**Example**

```
CLIPS> (clear)
CLIPS> (defmodule A (export deftemplate foo bar))
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (deftemplate A::bar (slot y))
CLIPS> (deffacts A::info (foo (x 3)) (bar (y 4)))
CLIPS> (defmodule B (import A deftemplate foo))
CLIPS> (reset)
CLIPS> (facts A)
f-1     (foo (x 3))
f-2     (bar (y 4))
For a total of 2 facts.
CLIPS> (facts B)
f-1     (foo (x 3))
For a total of 1 fact.
CLIPS>
```

Preview from Notesale.co.uk
Page 168 of 428

### 10.5.1 Specifying Instance-Names

Instance-names are required to be unique within a particular module, but multiple instances of the same name may be in scope at any one time. The syntax of instance-names has been extended to allow module specifications (note that the left and right brackets in bold are to be typed and do not indicate an optional part of the syntax).

**Syntax**

```
<instance-name> ::= [<symbol>] |
                    [::<symbol>] |
                    [<module>::symbol>]
```

Specifying just a symbol as the instance-name, such as [Rolls-Royce], will search for the instance in the current module only. Specifying only the :: before the name, such as [::Rolls-Royce], will search for the instance first in the current module and then recursively in the imported modules as defined in the module definition. Specifying both a symbol and a module name, such as [CARS::Rolls-Royce], searches for the instance only in the specified module. Regardless of which format is specified, the class of the instance must be in scope of the current module in order for the instance to be found.

## 10.6 MODULES AND RULE EXECUTION

Each module has its own pattern-matching network for its rules and its own agenda. When a **run** command is given, the agenda of the module which is the current focus is executed (note that the

INSTANCE for this attribute is equivalent to using both INSTANCE-NAME and INSTANCE-ADDRESS. ?VARIABLE allows any type to be stored.

## 11.2 ALLOWED CONSTANT ATTRIBUTES

The allowed constant attributes allow the constant values of a specific type which can be stored in a slot to be restricted. The list of values provided should either be a list of constants of the specified type or the keyword ?VARIABLE which means any constant of that type is allowed. The allowed-values attribute allows the slot to be restricted to a specific set of values (encompassing all types). Note the difference between using the attribute (allowed-symbols red green blue) and (allowed-values red green blue). The allowed-symbols attribute states that if the value is of type symbol, then its value must be one of the listed symbols. The allowed-values attribute completely restricts the allowed values to the listed values. The allowed-classes attribute does not restrict the slot value in the same manner as the other allowed constant attributes. Instead, if this attribute is specified and the slot value is either an instance address or instance name, then the class to which the instance belongs must be a class specified in the allowed-classes attribute or be a subclass of one of the specified classes.

Preview from Notesale.co.uk
Page 172 of 428

**Syntax**

```
<allowed-constant-attribute>
               ::= (allowed-symbols <symbol-list>) |
                   (allowed-strings <string-list>) |
                   (allowed-lexemes <lexeme-list>) |
                   (allowed-integers <integer-list>) |
                   (allowed-floats <float-list>) |
                   (allowed-numbers <number-list>) |
                   (allowed-instance-names <instance-list>) |
                   (allowed-classes <class-name-list>)
                   (allowed-values <value-list>)

<symbol-list>  ::= <symbol>+ | ?VARIABLE

<string-list>  ::= <string>+ | ?VARIABLE

<lexeme-list>  ::= <lexeme>+ | ?VARIABLE

<integer-list> ::= <integer>+ | ?VARIABLE

<float-list>   ::= <float>+ | ?VARIABLE

<number-list>  ::= <number>+ | ?VARIABLE

<instance-name-list>  ::= <instance-name>+ | ?VARIABLE

<class-name-list>  ::= <class-name>+ | ?VARIABLE

<value-list>   ::= <constant>+ | ?VARIABLE
```

Specifying the allowed-lexemes attribute is equivalent to specifying constant restrictions on both symbols and strings. A string or symbol must match one of the constants in the attribute list.

```
(defrule error
   (foo (x $?x))
   (bar (y $?y))
   (woz (z $?x $?y))
   =>)
CLIPS>
```

The variable ?x, found in the first pattern, can have a maximum of two fields. The variable ?y, found in the second pattern, can have a maximum of three fields. Added together, both variables have a maximum of five fields. Since slot z in the the third pattern has a minimum cardinality of seven, the variables ?x and ?y cannot satisfy the minimum cardinality restriction for this slot.

**Example 3**

```
CLIPS> (deftemplate foo (slot x (type SYMBOL)))
CLIPS>
(defrule error
   (foo (x ?x))
   (test (> ?x 10))
   =>)

[RULECSTR2] Previous variable bindings of ?x caused the type restriction for
argument #1 of the expression (> ?x 10)
found in CE #2 to be violated

ERROR:
(defrule error
   (foo (x ?x))
   (test (> ?x 10))
   =>)
CLIPS>
```

Preview from Notesale.co.uk
Page 176 of 428

The variable ?x, found in slot x of the first pattern, must be a symbol. Since the **>** function expects numeric values for its arguments, an error occurs.

The **build** function is not available for binary-load only or run-time CLIPS configurations (see the *Advanced Programming Guide*).

**Example**

```
CLIPS> (clear)
CLIPS> (build "(defrule foo (a) => (assert (b)))")
TRUE
CLIPS> (rules)
foo
For a total of 1 rule.
CLIPS>
```

### 12.3.7 Converting a String to Uppercase

The **upcase** function will return a string or symbol with uppercase alphabetic characters.

**Syntax**

```
(upcase <string-or-symbol-expression>)
```

**Example**

```
CLIPS> (upcase "This is a test of upcase")
"THIS IS A TEST OF UPCASE"
CLIPS> (upcase A_Word_Test_for_Upcase)
A_WORD_TEST_FOR_UPCASE
CLIPS>
```

Preview from Notesale.co.uk
Page 192 of 428

### 12.3.8 Converting a String to Lowercase

The **lowcase** function will return a string or symbol with lowercase alphabetic characters.

**Syntax**

```
(lowcase <string-or-symbol-expression>)
```

**Example**

```
CLIPS> (lowcase "This is a test of lowcase")
"this is a test of lowcase"
CLIPS> (lowcase A_Word_Test_for_Lowcase)
a_word_test_for_lowcase
CLIPS>
```

### 12.3.9 Comparing Two Strings

The **str-compare** function will compare two strings to determine their logical relationship (i.e., equal to, less than, greater than). The comparison is performed character-by-character until the strings are exhausted (implying equal strings) or unequal characters are found. The positions of the unequal characters within the ASCII character set are used to determine the logical relationship of unequal strings.

symbol, a number, or a string. Several logical names are predefined by CLIPS and are used extensively throughout the CLIPS code. These are

| Name | Description |
|---|---|
| **stdin** | The default for all user inputs. The read and readline functions read from stdin if t is specified as the logical name. |
| **stdout** | The default for all user outputs. The format and printout functions send output to stdout if t is specified as the logical name. |
| **wclips** | The CLIPS prompt is sent to this logical name. |
| **wdialog** | All informational messages are sent to this logical name. |
| **wdisplay** | Requests to display CLIPS information, such as facts or rules, are sent to this logical name. |
| **werror** | All error messages are sent to this logical name. |
| **wwarning** | All warning messages are sent to this logical name. |
| **wtrace** | All watch information is sent to this logical name (with the exception of compilations which is sent to wdialog). |

Any of these logical names may be used anywhere a logical name is expected.

Preview from Notesale.co.uk
Page 195 of 428

### 12.4.2 Common I/O Functions

CLIPS provides some of the most commonly needed I/O capabilities through several predefined functions.

#### 12.4.2.1 Open

The **open** function allows a user to open a file from the RHS of a rule and attaches a logical name to it. This function takes three arguments: (1) the name of the file to be opened; (2) the logical name which will be used by other CLIPS I/O functions to access the file; and (3) an optional mode specifier. The mode specifier must be one of the following strings:

| Mode | Means |
|---|---|
| **"r"** | read access only |
| **"w"** | write access only |
| **"r+"** | read and write access |
| **"a"** | append access only |
| **"wb"** | binary write access |

If the mode is not specified, a default of read access only is assumed. The access mode may not be meaningful in some operating systems.

**Syntax**

```
(open <file-name> <logical-name> [<mode>])
```

The <file-name> must either be a string or symbol and may include directory specifiers. If a string is used, the backslash (\) and any other special characters that are part of <file-name> must be escaped with a backslash. The logical name should not have been used previously. The **open** function returns TRUE if it was successful, otherwise FALSE.

**Example**

```
CLIPS> (open "myfile.clp" writeFile "w")
TRUE
CLIPS> (open "MS-DOS\\directory\\file.clp" readFile)
TRUE
CLIPS>
```

### 12.4.2.2 Close

The **close** function closes a file stream previously opened with the **open** command. The file is specified by a logical name previously attached to the desired stream.

Preview from Notesale.co.uk
Page 196 of 428

**Syntax**

```
(close [<logical-name>])
```

If **close** is called without arguments, all open files will be closed. The user is responsible for closing all files opened during execution. If files are not closed, the contents are not guaranteed correct, however, CLIPS will attempt to close all open files when the **exit** command is executed. The **close** function returns TRUE if any files were successfully closed, otherwise FALSE.

**Example**

```
CLIPS> (open "myfile.clp" writeFile "w")
TRUE
CLIPS> (open "MS-DOS\\directory\\file.clp" readFile)
TRUE
CLIPS> (close writeFile)
TRUE
CLIPS> (close writeFile)
FALSE
CLIPS> (close)
TRUE
CLIPS> (close)
FALSE
CLIPS>
```

where the first argument to bind, <variable>, is the local or global variable to be bound (it *may* have been bound previously). The bind function may also be used within a message-handler's body to set a slot's value.

If no <expression> is specified, then local variables are unbound and global variables are reset to their original value. If one <expression> is specified, then the value of <variable> is set to the return value from evaluating <expression>. If more than one <expression> is specified, then all of the <expressions> are evaluated and grouped together as a multifield value and the resulting value is stored in <variable>.

The bind function returns the symbol FALSE when a local variable is unbound, otherwise, the return value is the value to which <variable> is set.

**Example 1**

```
CLIPS> (defglobal ?*x* = 3.4)
CLIPS> ?*x*
3.4
CLIPS> (bind ?*x* (+ 8 9))
17
CLIPS> ?*x*
17
CLIPS> (bind ?*x* (create$ a b c d))
(a b c d)
CLIPS> ?*x*
(a b c d)
CLIPS> (bind ?*x* d e f)
(d e f)
CLIPS> ?*x*
(d e f)
CLIPS> (bind ?*x*)
3.4
CLIPS> ?*x*
3.4
CLIPS>
```

Preview from Notesale.co.uk
Page 214 of 428

**Example 2**

```
CLIPS>
(defclass A (is-a USER)
   (role concrete)
   (slot x) (slot y))
CLIPS>
(defmessage-handler A init after ()
   (bind ?self:x 3)
   (bind ?self:y 4))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print)
[a] of A
(x 3)
(y 4)
CLIPS>
```

```
  (slot x)
  (multislot y (cardinality ?VARIABLE 5))
  (multislot z (cardinality 3 ?VARIABLE)))
CLIPS> (deftemplate-slot-cardinality A y)
(0 5)
CLIPS> (deftemplate-slot-cardinality A z)
(3 +oo)
CLIPS>
```

### 12.8.4 Testing whether a Deftemplate Slot has a Default

This function returns the symbol static if the specified slot in the specified deftemplate has a static default (whether explicitly or implicitly defined), the symbol dynamic if the slot has a dynamic default, or the symbol FALSE if the slot does not have a default. An error is generated if the specified deftemplate or slot does not exist.

**Syntax**

```
(deftemplate-slot-defaultp <deftemplate-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(deftemplate A
   (slot w)
   (slot x (default ?NONE))
   (slot y (default 1))
   (slot z (default-dynamic (gensym*))))
CLIPS> (deftemplate-slot-defaultp A w)
static
CLIPS> (deftemplate-slot-defaultp A x)
FALSE
CLIPS> (deftemplate-slot-defaultp A y)
static
CLIPS> (deftemplate-slot-defaultp A z)
dynamic
CLIPS>
```

Preview from Notesale.co.uk
Page 226 of 428

### 12.8.5 Getting the Default Value for a Deftemplate Slot

This function returns the default value associated with a deftemplate slot. If a slot has a dynamic default, the expression will be evaluated when this function is called. The symbol FALSE is returned if an error occurs.

**Syntax**

```
(deftemplate-slot-default-value <deftemplate-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(deftemplate A
```

```
  (slot x (default 3))
  (multislot y (default a b c))
  (slot z (default-dynamic (gensym))))
CLIPS> (deftemplate-slot-default-value A x)
3
CLIPS> (deftemplate-slot-default-value A y)
(a b c)
CLIPS> (deftemplate-slot-default-value A z)
gen1
CLIPS> (deftemplate-slot-default-value A z)
gen2
CLIPS>
```

### 12.8.6 Deftemplate Slot Existence

This function returns the symbol TRUE if the specified slot is present in the specified deftemplate, FALSE otherwise.

**Syntax**

```
(deftemplate-slot-existp <deftemplate-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS> (deftemplate A (slot x))
CLIPS> (deftemplate-slot-existp A x)
TRUE
CLIPS> (deftemplate-slot-existp A y)
FALSE
CLIPS>
```

Preview from Notesale.co.uk
Page 227 of 428

### 12.8.7 Testing whether a Deftemplate Slot is a Multifield Slot

This function returns the symbol TRUE if the specified slot in the specified deftemplate is a multifield slot. Otherwise, it returns the symbol FALSE. An error is generated if the specified deftemplate or slot does not exist.

**Syntax**

```
(deftemplate-slot-multip <deftemplate-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS> (deftemplate A (slot x) (multislot y))
CLIPS> (deftemplate-slot-multip A x)
FALSE
CLIPS> (deftemplate-slot-multip A y)
TRUE
CLIPS>
```

```
CLIPS> (facts)
f-0     (example fact)
For a total of 1 fact.
CLIPS> (fact-existp 0)
TRUE
CLIPS> (retract 0)
CLIPS> (fact-existp ?*x*)
FALSE
CLIPS>
```

### 12.9.8 Determining the Deftemplate (Relation) Name Associated with a Fact

The **fact-relation** function returns the deftemplate (relation) name associated with the fact. FALSE is returned if the specified fact does not exist.

**Syntax**

```
(fact-relation <fact-address-or-index>)
```

**Example**

```
CLIPS> (clear)
CLIPS> (assert (example fact))
<Fact-0>
CLIPS> (fact-relation 0)
example
CLIPS>
```

Preview from Notesale.co.uk
Page 234 of 428

### 12.9.9 Determining the Slot Names Associated with a Fact

The **fact-slot-names** function returns the slot names associated with the fact in a multifield value. The symbol *implied* is returned for an ordered fact (which has a single implied multifield slot). FALSE is returned if the specified fact does not exist.

**Syntax**

```
(fact-slot-names <fact-address-or-index>)
```

**Example**

```
CLIPS> (clear)
CLIPS> (deftemplate foo (slot bar) (multislot yak))
CLIPS> (assert (foo (bar 1) (yak 2 3)))
<Fact-0>
CLIPS> (fact-slot-names 0)
(bar yak)
CLIPS> (assert (another a b c))
<Fact-1>
CLIPS> (fact-slot-names 1)
(implied)
CLIPS>
```

### 12.10.2 Determining the Module in which a Deffacts is Defined

This function returns the module in which the specified deffacts name is defined.

**Syntax**

```
(deffacts-module <deffacts-name>)
```

## 12.11 DEFRULE FUNCTIONS

The following functions provide ancillary capabilities for the defrule construct.

### 12.11.1 Getting the List of Defrules

The function **get-defrule-list** returns a multifield value containing the names of all defrule constructs visible to the module specified by <module-name> or to the current module if none is specified. If * is specified as the module name, then all defrules are returned.

**Syntax**

```
(get-defrule-list)
```

**Example**

```
CLIPS> (clear)
CLIPS> (get-defrule-list)
()
CLIPS> (defrule foo =>)
CLIPS> (defrule bar =>)
CLIPS> (get-defrule-list)
(foo bar)
CLIPS>
```

Preview from Notesale.co.uk
Page 245 of 428

### 12.11.2 Determining the Module in which a Defrule is Defined

This function returns the module in which the specified defrule name is defined.

**Syntax**

```
(defrule-module <defrule-name>)
```

## 12.12 AGENDA FUNCTIONS

The following functions provide ancillary capabilities manipulating the agenda.

```
      .
<mth-restriction-class-count>
<mth-restriction-first-class>
      .
      .
      .
<mth-restriction-nth-class>
```

**Syntax**

```
(get-method-restrictions <generic-function-name>
                         <method-index>)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(defmethod foo 50 ((?a INTEGER SYMBOL) (?b (= 1 1)) $?c))
CLIPS> (get-method-restrictions foo 50)
(2 -1 3 7 11 13 FALSE 2 INTEGER SYMBOL TRUE 0 FALSE 0)
CLIPS>
```

## 12.16 CLIPS OBJECT-ORIENTED LANGUAGE (COOL) FUNCTIONS

The following functions provide ancillary capabilities for COOL.

Preview from Notesale.co.uk
Page 253 of 428

### 12.16.1 Class Functions

#### 12.16.1.1 Getting the List of Defclasses

The function **get-defclass-list** returns a multifield value containing the names of all defclass constructs visible to the module specified by <module-name> or to the current module if none is specified. If * is specified as the module name, then all defclasses are returned.

**Syntax**

```
(get-defclass-list [<module-name>])
```

**Example**

```
CLIPS> (clear)
CLIPS> (get-defclass-list)
(FLOAT INTEGER SYMBOL STRING MULTIFIELD EXTERNAL-ADDRESS FACT-ADDRESS INSTANCE-
ADDRESS INSTANCE-NAME OBJECT PRIMITIVE NUMBER LEXEME ADDRESS INSTANCE USER
INITIAL-OBJECT)
CLIPS> (defclass FOO (is-a USER))
CLIPS> (defclass BAR (is-a USER))
CLIPS> (get-defclass-list)
(FLOAT INTEGER SYMBOL STRING MULTIFIELD EXTERNAL-ADDRESS FACT-ADDRESS INSTANCE-
ADDRESS INSTANCE-NAME OBJECT PRIMITIVE NUMBER LEXEME ADDRESS INSTANCE USER
INITIAL-OBJECT FOO BAR)
CLIPS>
```

**Example**

```
CLIPS> (class-subclasses PRIMITIVE)
(NUMBER LEXEME MULTIFIELD EXTERNAL-ADDRESS)
CLIPS> (class-subclasses PRIMITIVE inherit)
(NUMBER INTEGER FLOAT LEXEME SYMBOL STRING MULTIFIELD ADDRESS EXTERNAL-ADDRESS
FACT-ADDRESS INSTANCE-ADDRESS INSTANCE INSTANCE-NAME)
CLIPS>
```

### 12.16.1.16 Getting the List of Slots for a Class

This function groups the names of the explicitly defined slots of a class into a multifield variable. If the optional argument "inherit" is given, inherited slots are also included. A multifield of length zero is returned if an error occurs.

**Syntax**

```
(class-slots <class-name> [inherit])
```

**Example**

```
CLIPS> (defclass A (is-a USER) (slot x))
CLIPS> (defclass B (is-a A) (slot y))
CLIPS> (class-slots B)
(y)
CLIPS> (class-slots B inherit)
(x y)
CLIPS>
```

Preview from Notesale.co.uk
Page 257 of 428

### 12.16.1.17 Getting the List of Message-Handlers for a Class

This function groups the class names, message names and message types of the message-handlers attached direct to class into a multifield variable (implicit slot-accessors are not included). If the optional argument "inherit" is given, inherited message-handlers are also included. A multifield of length zero is returned if an error occurs.

**Syntax**

```
(get-defmessage-handler-list <class-name> [inherit])
```

**Example**

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
CLIPS> (defmessage-handler A foo ())
CLIPS> (get-defmessage-handler-list A)
(A foo primary)
CLIPS> (get-defmessage-handler-list A inherit)
(USER init primary USER delete primary USER create primary USER print primary USER
direct-modify primary USER message-modify primary USER direct-duplicate primary
USER message-duplicate primary A foo primary)
CLIPS>
```

```
(0 5)
CLIPS> (slot-cardinality A z)
(3 +oo)
CLIPS>
```

### 12.16.1.22 Getting the Allowed Values for a Slot

This function groups the allowed values for a slot (specified in any of allowed-… facets for the slots) into a multifield variable. If no allowed-… facets were specified for the slot, then the symbol FALSE is returned. A multifield of length zero is returned if an error occurs.

**Syntax**

```
(slot-allowed-values <class-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot x)
  (slot y (allowed-integers 2 3) (allowed-symbols foo)))
CLIPS> (slot-allowed-values A x)
FALSE
CLIPS> (slot-allowed-values A y)
(2 3 foo)
CLIPS>
```

Preview from Notesale.co.uk
Page 260 of 428

### 12.16.1.23 Getting the Numeric Range for a Slot

This function groups the minimum and maximum numeric ranges allowed a slot into a multifield variable. A minimum value of infinity is indicated by the symbol **-oo** (the minus character followed by two lowercase o's—not zeroes). A maximum value of infinity is indicated by the symbol **+oo** (the plus character followed by two lowercase o's—not zeroes). The symbol FALSE is returned for slots in which numeric values are not allowed. A multifield of length zero is returned if an error occurs.

**Syntax**

```
(slot-range <class-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot x)
  (slot y (type SYMBOL))
  (slot z (range 3 10)))
CLIPS> (slot-range A x)
(-oo +oo)
CLIPS> (slot-range A y)
FALSE
```

```
CLIPS> (slot-range A z)
(3 10)
CLIPS>
```

### 12.16.1.24 Getting the Default Value for a Slot

This function returns the default value associated with a slot. If a slot has a dynamic default, the expression will be evaluated when this function is called. The symbol FALSE is returned if an error occurs.

**Syntax**

```
(slot-default-value <class-name> <slot-name>)
```

**Example**

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot x (default 3))
  (multislot y (default a b c))
  (slot z (default-dynamic (gensym))))
CLIPS> (slot-default-value A x)
3
CLIPS> (slot-default-value A y)
(a b c)
CLIPS> (slot-default-value A z)
gen1
CLIPS> (slot-default-value A z)
gen2
CLIPS>
```

Preview from Notesale.co.uk
Page 261 of 428

### 12.16.1.25 Setting the Defaults Mode for Classes

This function sets the defaults mode used when classes are defined. The old mode is the return value of this function.

**Syntax**

```
(set-class-defaults-mode <mode>)
```

where <mode> is either convenience or conservation. By default, the class defaults mode is convenience. If the mode is convenience, then for the purposes of role inheritance, system defined class behave as concrete classes; for the purpose of pattern-match inheritance, system defined classes behave as reactive classes unless the inheriting class is abstract; and the default setting for the create-accessor facet of the class' slots is read-write. If the class defaults mode is conservation, then the role and reactivity of system-defined classes is unchanged for the purposes of role and pattern-match inheritance and the default setting for the create-accessor facet of the class' slots is ?NONE.

### 12.16.2.2 Calling Shadowed Handlers

If the conditions are such that the function **next-handlerp** would return the symbol TRUE, then calling this function will execute the shadowed method. Otherwise, a message execution error (see section 9.5.4) will occur. In the event of an error, the return value of this function is the symbol FALSE, otherwise it is the return value of the shadowed handler. The shadowed handler is passed the same arguments as the calling handler.

A handler may continue execution after calling **call-next-handler**. In addition, a handler may make multiple calls to **call-next-handler**, and the same shadowed handler will be executed each time.

**Syntax**

```
(call-next-handler)
```

**Example**

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS>
(defmessage-handler A print-args ($?any)
  (printout t "A: " ?any crlf)
  (if (next-handlerp) then
     (call-next-handler)))
CLIPS>
(defmessage-handler USER print-args ($?any)
  (printout t "USER: " ?any crlf))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print-args 1 2 3 4)
A: (1 2 3 4)
USER: (1 2 3 4)
CLIPS>
```

Preview from Notesale.co.uk
Page 263 of 428

### 12.16.2.3 Calling Shadowed Handlers with Different Arguments

This function is identical to **call-next-handler** except that this function can change the arguments passed to the shadowed handler.

**Syntax**

```
(override-next-handler <expression>*)
```

**Example**

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS>
(defmessage-handler A print-args ($?any)
  (printout t "A: " ?any crlf)
  (if (next-handlerp) then
     (override-next-handler (rest$ ?any))))
```

**Syntax**

```
(apropos <lexeme>)
```

**Example**

```
CLIPS> (apropos pen)
dependents
mv-append
open
dependencies
CLIPS>
```

## 13.2 DEBUGGING COMMANDS

The following commands control the CLIPS debugging features.

### 13.2.1 Generating Trace Files

Sends all information normally sent to the logical names **wclips**, **wdialog**, **wdisplay**, **werror**, **wwarning**, **wtrace**, and **stdout** to <file-name> as well as to their normal destination. Additionally, all information received from logical name **stdin** is also sent to <file-name> as well as being returned by the requesting function. This function returns TRUE if the dribble file was successfully opened, otherwise FALSE is returned.

Preview from Notesale.co.uk
Page 281 of 428

**Syntax**

```
(dribble-on <file-name>)
```

### 13.2.2 Closing Trace Files

Stops sending trace information to the dribble file. This function returns TRUE if the dribble file was successfully closed, otherwise FALSE is returned.

**Syntax**

```
(dribble-off)
```

### 13.2.3 Enabling Watch Items

This function causes messages to be displayed when certain CLIPS operations take place.

**Syntax**

```
(watch <watch-item>)

<watch-item> ::= all |
                 compilations |
                 statistics |
                 focus |
```

This command displays the current state of all watch items. If called without the <watch-item> argument, the global watch state of all watch items is displayed. If called with the <watch-item> argument, the global watch state for that item is displayed followed by the individual watch states for each item of the specified type which can be watched. This function has no return value.

**Example**

```
CLIPS> (list-watch-items)
facts = off
instances = off
slots = off
rules = off
activations = off
messages = off
message-handlers = off
generic-functions = off
methods = off
deffunctions = off
compilations = on
statistics = off
globals = off
focus = off
CLIPS> (list-watch-items facts)
facts = off
MAIN:
   initial-fact = off
CLIPS>
```

Preview from Notesale.co.uk
Page 284 of 428

## 13.3 DEFTEMPLATE COMMANDS

The following commands manipulate deftemplates.

### 13.3.1 Displaying the Text of a Deftemplate

Displays the text of a given deftemplate. This function has no return value.

**Syntax**

```
(ppdeftemplate <deftemplate-name>)
```

### 13.3.2 Displaying the List of Deftemplates

Displays the names of all deftemplates. This function has no return value.

**Syntax**

```
(list-deftemplates [<module-name>])
```

If <module-name> is unspecified, then the names of all deftemplates in the current module are displayed. If <module-name> is specified, then the names of all deftemplates in the specified

## 13.7 AGENDA COMMANDS

The following commands manipulate agenda.

### 13.7.1 Displaying the Agenda

Displays all activations on the agenda. This function has no return value.

**Syntax**

```
(agenda [<module-name>])
```

If <module-name> is unspecified, then all activations in the current module (not the current focus) are displayed. If <module-name> is specified, then all activations on the agenda of the specified module are displayed. If <module-name> is the symbol *, then the activations on all agendas in all modules are displayed.

### 13.7.2 Running CLIPS

Starts execution of the rules. If the optional first argument is positive, execution will cease after the specified number of rule firings or when the agenda contains no rule activations. If there are no arguments or the first argument is a negative integer, execution will cease when the agenda contains no rule activations. If the focus stack is empty, then the MAIN module is automatically becomes the current focus. The **run** command has no additional effect if evaluated while rules are executing. Note that the number of rules fired and timing information is no longer printed after the completion of the run command unless the statistics item is being watched (see section 13.2). If the rules item is being watched, then an informational message will be printed each time a rule is fired. This function has no return value.

Preview from Notesale.co.uk
Page 294 of 428

**Syntax**

```
(run [<integer-expression>])
```

### 13.7.3 Focusing on a Group of Rules

Pushes one or more modules onto the focus stack. The specified modules are pushed onto the focus stack in the reverse order they are listed. The current module is set to the last module pushed onto the focus stack. The current focus is the top module of the focus stack. Thus (focus A B C) pushes C, then B, then A unto the focus stack so that A is now the current focus. Note that the current focus is different from the current module. Focusing on a module implies “remembering” the current module so that it can be returned to later. Setting the current module with the **set-current-module** function changes it without remembering the old module. Before a rule executes, the current module is changed to the module in which the executing rule is defined (the current focus). This function returns a boolean value: FALSE if an error occurs, otherwise TRUE.

**Syntax**

```
(mem-requests)
```

### 13.13.3 Releasing Memory Used by CLIPS

Releases all free memory held internally by CLIPS back to the operating system. CLIPS will automatically call this function if it is running low on memory to allow the operating system to coalesce smaller memory blocks into larger ones. This function generally should not be called unless the user knows *exactly* what he/she is doing (since calling this function can prevent CLIPS from reusing memory efficiently and thus slow down performance). This function returns an integer representing the amount of memory freed to the operating system.

**Syntax**

```
(release-mem)
```

### 13.13.4 Conserving Memory

Turns on or off the storage of information used for **save** and pretty print commands. This can save considerable memory in a large system. It should be called *prior* to loading any constructs. This function has no return value.

Preview from Notesale.co.uk
Page 312 of 428

**Syntax**

```
(conserve-mem <value>)
```

where value is either **on** or **off**.

## 13.14 ON-LINE HELP SYSTEM

CLIPS provides an on-line help facility for use from the top-level interface. The help system uses CLIPS' external text manipulation capabilities (see section 13.15). Thus, it is possible to add or change entries in the help file or to construct new help files with information specific to the user's system.

### 13.14.1 Using the CLIPS Help Facility

The **help** facility displays menus of topics and prompts the user for a choice. It then references the help file for that information. The **help** facility can be called with or without a command-line topic.

**Syntax**

```
(help [<path>])
```

Preview from Notesale.co.uk
Page 324 of 428

- Metrowerks CodeWarrior 9.6 for Mac OS X.
- Xcode 2.3 for Mac OS X.
- Microsoft Visual C++ .NET 2003 for Windows.

## B.3 VERSION 6.23

- **Fact-Set Query Functions** – Six new functions similar to the instance set query functions have been added for determining and performing actions on sets of facts that satisfy user-defined queries (see section 12.9.12): any-factp, find-fact, find-all-facts, do-for-fact, do-for-all-facts, and delayed-do-for-all-facts. The GetNextFactInTemplate function (see section 4.4.17 of the *Advanced Programming Guide*) allows iteration from C over the facts belonging to a specific deftemplate.

- **Bug Fixes** - The following bugs were fixed by the 6.23 release:
  - Passing the wrong number of arguments to a deffunction through the funcall function could cause unpredictable behavior including memory corruption.
  - A large file name (at least 60 characters) passed into the fetch command causes a buffer overrun.
  - A large file name (at least 60 characters) passed into the constructs-to-c command causes a buffer overrun.
  - A large defclass or defgeneric name (at least 500 characters) causes a buffer overrun when the profile-info command is called.
  - A large module or construct name (at least 500 characters) causes a buffer overrun when the get-<construct>-list command is called.
  - The FalseSymbol and TrueSymbol constants were not defined as described in the *Advanced Programming Guide*. These constants have have now been defined as macros so that their corresponding environment companion functions (EnvFalseSymbol and EnvTrueSymbol) could be defined. See the *Advanced Programming Guide* for more details.
  - The slot-writablep function returns TRUE for slots having initialize-only access.

Preview from Notesale.co.uk
Page 328 of 428

**conditional element** — A restriction on the LHS of a rule which must be satisfied in order for the rule to be applicable (also referred to as a CE).

**conflict resolution strategy** — A method for determining the order in which rules should fire among rules with the same salience. There are seven different conflict resolution strategies: depth, breadth, simplicity, complexity, lex, mea, and random.

**consequent** — The RHS of a rule.

**constant** — A non-varying single field value directly expressed as a series of characters.

**constraint** — In patterns, a constraint is a requirement that is placed on the value of a field from a fact or instance that must be satisified in order for the pattern to be satisfied. For example, the ~red constraint is satisfied if the field to which the constraint is applied is not the symbol *red*. The term constraint is also used to refer to the legal values allowed in the slots of facts and instances.

Preview from Notesale.co.uk
Page 342 of 428

**construct** — A high level CLIPS abstraction used to add components to the knowledge base.

**current focus** — The module from which activations are selected to be fired.

**current module** — The module to which newly defined constructs that do not have a module specifier are added. Also is the default module for certain commands which accept as an optional argument a module name (such as list-defrules).

**daemon** — A message-handler which executes implicitly whenever some action is taken upon an object, such as initialization, deletion, or slot access.

**deffunction** — A non-overloaded function written directly in CLIPS.

**deftemplate fact** — A deftemplate name followed by a list of named fields (slots) and specific values used to represent a deftemplate object. Note that a deftemplate fact has no inheritance. Also called a non-ordered fact.

**deftemplate object** — An informal term for the entity described by a deftemplate. A

character “f”, followed by a dash, followed by the fact-index of the fact.

**fact-index** A unique integer index used to identify a particular fact.

**fact-list** The list of current facts.

**field** A placeholder (named or unnamed) that has a value.

**fire** A rule is said to have fired if all of its conditions are satisfied and the actions then are executed.

**float** A number that begins with an optional sign followed optionally in order by zero or more digits, a decimal point, zero or more digits, and an exponent (consisting of an e or E followed by an integer). A floating point number must have at least one digit in it (not including the exponent) and must either contain a decimal point or an exponent (see section 2.3.1 for more details).

**focus** As a verb, refers to changing the current focus. As a noun, refers to the current focus.

**focus stack** The list of modules that have been focused upon. The module at the top of the focus stack is the current focus. When all the activations from the current focus have been fired, the current focus is removed from the focus stack and the next module on the stack becomes the current focus.

**function** A piece of executable code identified by a specific name which returns a useful value or performs a useful side effect. Typically only used to refer to functions which do return a value (whereas commands and actions are used to refer to functions which do not return a value).

**generic dispatch** The process whereby applicable methods are selected and executed for a particular generic function call.

**generic function** A function written in CLIPS which can do different things depending on what the number and types of its arguments.

**inference engine** The mechanism provided by CLIPS which automatically matches patterns against the current state of the fact-list and list of instances and determines which rules are applicable.

Preview from Notesale.co.uk
Page 344 of 428

# Appendix E - Performance Considerations

This appendix explains various techniques that the user can apply to a CLIPS program to maximize performance. Included are discussions of pattern ordering in rules, use of deffunctions in lieu of non-overloaded generic functions, parameter restriction ordering in generic function methods, and various approaches to improving the speed of message-passing and reading slots of instances.

## E.1 ORDERING OF PATTERNS ON THE LHS

The issues which affect performance of a rule-based system are considerably different from those which affect conventional programs. This section discusses the single most important issue: the ordering of patterns on the LHS of a rule.

CLIPS is a rule language based on the RETE algorithm. The RETE algorithm was designed specifically to provide very efficient pattern-matching. CLIPS has attempted to implement this algorithm in a manner that combines efficient performance with powerful features. When used properly, CLIPS can provide very reasonable performance, even on microcomputers. However, to use CLIPS properly requires some understanding of how the pattern-matcher works.

Preview from Notesale.co.uk
Page 353 of 428

Prior to initiating execution, each rule is loaded into the system and a network of all patterns that appear on the LHS of any rule is constructed. As facts and instances of reactive classes (referred to collectively as pattern entities) are created, they are filtered through the pattern network. If the pattern entities match any of the patterns in the network, the rules associated with those patterns are partially instantiated. When pattern entities exist that match all patterns on the LHS of the rule, variable bindings (if any) are considered. They are considered from the top to the bottom; i.e., the first pattern on the LHS of a rule is considered, then the second, and so on. If the variable bindings for all patterns are consistent with the constraints applied to the variables, the rules are activated and placed on the agenda.

This is a very simple description of what occurs in CLIPS, but it gives the basic idea. A number of important considerations come out of this. Basic pattern-matching is done by filtering through the pattern network. The time involved in doing this is fairly constant. The slow portion of basic pattern-matching comes from comparing variable bindings across patterns. Therefore, the single most important performance factor is the ordering of patterns on the LHS of the rule. Unfortunately, there are no hard and fast methods that will always order the patterns properly. At best, there seem to be three “quasi” methods for ordering the patterns.

1) Most specific to most general. The more wildcards or unbound variables there are in a pattern, the lower it should go. If the rule firing can be controlled by a single pattern, place that pattern first. This technique often is used to provide control structure in an expert system; e.g., some kind of “phase” fact. Putting this kind of pattern first will *guarantee* that

Preview from Notesale.co.uk
Page 356 of 428

Example:

```
CLIPS> (defrule foo (a ~?x) =>)
```

**[ARGACCES1] Function <name> expected at least <minimum> and no more than <maximum> argument(s)**
This error occurs when a function receives less than the minimum number or more than the maximum number of argument(s) expected.

**[ARGACCES2] Function <function-name> was unable to open file <file-name>**
This error occurs when the specified function cannot open a file.

**[ARGACCES3] Function <name1> received a request from function <name2> for argument #<number> which is non-existent**
This error occurs when a function is passed fewer arguments than were expected.

**[ARGACCES4] Function <name> expected exactly <number> argument(s)**
This error occurs when a function that expects a precise number of argument(s) receives an incorrect number of arguments.

**[ARGACCES4] Function <name> expected at least <number> argument(s)**
This error occurs when a function does not receive the minimum number of argument(s) that it expected.

**[ARGACCES4] Function <name> expected no more than <number> argument(s)**
This error occurs when a function receives more than the maximum number of argument(s) expected.

Preview from Notesale.co.uk
Page 360 of 428

**[ARGACCES5] Function <name> expected argument #<number> to be of type <data-type>**
This error occurs when a function is passed the wrong type of argument.

**[ARGACCES6] Function <name1> received a request from function <name2> for argument #<number> which is not of type <data-type>**
This error occurs when a function requests from another function the wrong type of argument, typically a **string** or **symbol**, when expecting a **number** or vice versa.

**[BLOAD1] Cannot load <construct type> construct with binary load in effect.**
If the bload command was used to load in a binary image, then the named construct cannot be entered until a clear command has been performed to remove the binary image.

**[BLOAD2] File <file-name> is not a binary construct file**
This error occurs when the bload command is used to load a file that was not created with the bsave command.

Example:
```
CLIPS> (deftemplate foo (slot x (type SYMBOL)))
CLIPS> (assert (foo (x 3)))
```

**[CSTRNPSR1] The <first attribute name> attribute conflicts with the <second attribute name> attribute.**
This error message occurs when two slot attributes conflict.

Example:
```
CLIPS> (deftemplate foo (slot x (type SYMBOL) (range 0 2)))
```

**[CSTRNPSR2] Minimum <attribute> value must be less than or equal to the maximum <attribute> value.**
The minimum attribute value for the range and cardinality attributes must be less than or equal to the maximum attribute value for the attribute.

Example:
```
CLIPS> (deftemplate foo (slot x (range 8 1)))
```

**[CSTRNPSR3] The <first attribute name> attribute cannot be used in conjunction with the <second attribute name> attribute.**
The use of some slot attributes excludes the use of other slot attributes.

Preview from Notesale.co.uk
Page 366 of 428

Example:
```
CLIPS> (deftemplate foo (slot x (allowed-values a)
                                (allowed-symbols b)))
```

**[CSTRNPSR4] Value does not match the expected type for the <attribute name> attribute.**
The arguments to an attribute must match the type expected for that attribute (e.g. integers must be used for the allowed-integers attribute).

Example:
```
CLIPS> (deftemplate example (slot x (allowed-integers 3.0)))
```

**[CSTRNPSR5] The cardinality attribute can only be used with multifield slots.**
The cardinality attribute can only be used for slots defined with the multislot keyword.

Example:
```
CLIPS> (deftemplate foo (slot x (cardinality 1 1)))
```

**[DEFAULT1] The default value for a single field slot must be a single field value**
This error occurs when the default or default-dynamic attribute for a single-field slot does not contain a single value or an expression returning a single value.

Example:
```
CLIPS> (deftemplate error (slot x (default)))
```

**[MSGPASS3] Static reference to slot <name> of class <name> does not apply to <instance-name> of <class-name>.**
This error occurs when a static reference to a slot in a superclass by a message-handler attached to that superclass is incorrectly applied to an instance of a subclass which redefines that slot. Static slot references always refer to the slot defined in the class to which the message-handler is attached.

Example:

```
CLIPS>
(defclass A (is-a USER)
  (slot foo))
CLIPS>
(defclass B (is-a A)
  (role concrete)
  (slot foo))
CLIPS>
(defmessage-handler A access-foo ()
  ?self:foo)
CLIPS> (make-instance b of B)
[b]
CLIPS> (send [b] access-foo)
```

**[MSGPSR1] A class must be defined before its message-handlers.**
A message-handler can only be attached to an existing class.

Example:

```
CLIPS> (defmessage-handler bogus-class foo ())
```

Preview from Notesale.co.uk
Page 386 of 428

**[MSGPSR2] Cannot (re)define message-handlers during execution of other message-handlers for the same class.**
No message-handlers for a class can be loaded while any current message-handlers attached to the class are executing.

Example:

```
CLIPS> (defclass A (is-a USER))
CLIPS> (make-instance a of A)
[a]
CLIPS>
(defmessage-handler A build-new ()
  (build "(defmessage-handler A new ())"))
CLIPS> (send [a] build-new)
```

**[MSGPSR3] System message-handlers may not be modified.**
There are four primary message-handlers attached to the class USER which cannot be modified: init, delete, create and print.

Example:

```
CLIPS> (defmessage-handler USER init ())
```

# Appendix I - Reserved Function Names

This appendix lists all of the functions provided by either standard CLIPS or various CLIPS extensions. They should be considered reserved function names, and users should not create user-defined functions with any of these names.

!=
*
**
+
-
/
<
<=
<>
=
>
>=
abs
acos
acosh
acot
acoth
acsc
acsch
active-duplicate-instance
active-initialize-instance
active-make-instance
active-message-duplicate-instance
active-message-modify-instance
active-modify-instance
agenda
and
any-instancep
apropos
asec
asech
asin
asinh
assert
assert-string
atan
atanh
batch
batch*
bind
bload
bload-instances
break
browse-classes
bsave
bsave-instances
build
call-next-handler
call-next-method
call-specific-method
class
class-abstractp
class-existp
class-reactivep
class-slots
class-subclasses
class-superclasses
clear
clear-focus-stack
close
conserve-mem
constructs-to-c
cos
cosh
cot
coth
create$
csc
csch
defclass-module
deffacts-module
deffunction-module

Preview from Notesale.co.uk
Page 405 of 428

Preview from Notesale.co.uk
Page 410 of 428

bsave-instances ..... **284**
build ..... **165**, 303
C ..... iii, 7, 9, 12, 15, 16, 21
call-next-handler ..... 82, 113, 114, **237**
call-next-method ..... 81, 82, 85, **224**, 225
call-specific-method ..... 75, 82, 85, **225**
carriage return ..... 7
case sensitive ..... 7
check-syntax ..... 167, 307
class ..... 8, **13**, 78, 240, 276, 278
  abstract ..... 87, **92**, 235, 276
  concrete ..... 88, **92**, 235
  existence ..... **227**
  immediate ..... **92**, 103
  non-reactive ..... **92**
  precedence ..... 90
  reactive ..... 88, **92**, 235
  specific ..... **90**, 92, 97, 113
  system ..... **87**
    ADDRESS ..... **87**
    EXTERNAL-ADDRESS ..... **87**
    FACT-ADDRESS ..... **87**
    FLOAT ..... **87**
    INITIAL-OBJECT ..... **87**
    INSTANCE ..... **87**
    INSTANCE-ADDRESS ..... **87**
    INSTANCE-NAME ..... **87**
    INTEGER ..... **87**
    LEXEME ..... **87**
    MULTIFIELD ..... **87**
    NUMBER ..... **87**
    OBJECT ..... **87**, 90, 278
    PRIMITIVE ..... **87**
    STRING ..... **87**
    SYMBOL ..... **87**
    USER ..... **87**, 90, 108, 119, 239, 283
  user-defined ..... 8, **282**
  *user-defined* ..... 13
class function ..... 223, **240**
class-abstractp ..... **229**
class-existp ..... **228**
class-reactivep ..... **230**
class-slots ..... **231**
class-subclasses ..... **230**
class-superclasses ..... **230**
clear 11, 25, 67, 118, 137, 139, 143, 250, **251**
clear-focus-stack ..... **270**
CLIPS ..... iii
CLIPSFunctionCall ..... 310, 311, 312
CLOS ..... 75, 87
close ..... **170**
command ..... 3, **151**, 249
command prompt ..... **3**
comment ..... 7, 10
Common Lisp Object System ..... iv
condition ..... **15**
conditional element ..... **15**, 25, 27, 33, 63
  and ..... 27, 33, **52**
  exists ..... 33, **54**
  forall ..... 33, **56**
  logical ..... 33, **58**
  not ..... 33, **53**
  or ..... 33, **51**
  pattern ..... 27, 33
  initial-fact ..... 27, 52, 53
  initial-object ..... 27, 52, 53
    literal ..... **34**
  test ..... 30, 33, **49**
conflict resolution strategy ... 15, **28**, 29, 251, 269
  breadth ..... **29**
  complexity ..... **30**
  depth ..... **29**
  lex ..... **30**
  mea ..... **31**
  random ..... **31**
  simplicity ..... **29**
consequent ..... **15**
conservation ..... 235
conserve-mem ..... 250, **286**
constant ..... 3, **8**
constraint ..... 33, **40**, 43
  connective ..... 34, **40**
  field ..... **33**
  literal ..... **34**
  predicate ..... 34, **43**, 49

Preview from Notesale.co.uk
Page 420 of 428

direct ..... 87, 88, **92**, 96
initialization ..... 108, 115, 118, 239
manipulation ..... **115**
printing ..... 110
instance-address ... 6, **8**, 9, 49, 171, 240, 241, 329
instance-addressp ..... **241**
instance-existp ..... 242
instance-list ..... 14, 27
instance-name ..... 6, **8**, 129, **240**, 241
instance-namep ..... 242
instance-name-to-symbol ..... **241**
instancep ..... **241**
instances ..... **282**
instance-set ..... **128**
action ..... 132
class restriction ..... **128**
distributed action ..... **131**
member ..... **128**
member variable ..... **129**, 132
query ..... 19, **130**, 132, 329
query execution error ..... 132
query functions ..... **133**
template ..... 128
integer ..... 8, **182**
integerp ..... **151**
integration ..... 5
Interfaces Guide ..... v, 3
-l ..... 5
left-hand side ..... **15**
length ..... **196**
length$ ..... 93, 105, **162**, 196
less than ..... 7
lexemep ..... **152**
LHS ..... **27**
line feed ..... 7
LISP ..... iii, 15
list-defclasses ..... 276
list-deffacts ..... **262**
list-deffunctions ..... **272**
list-defgenerics ..... **273**
list-defglobals ..... **271**
list-definstances ..... **282**
list-defmessage-handlers ..... **280**
list-defmethods ..... 77, 82, **274**, 275
list-defmodules ..... **285**
list-defrules ..... **263**
list-deftemplates ..... **258**
list-focus-stack ..... **269**
list-watch-items ..... **257**
load ..... 5, **249**, 250, 252
load* ..... **249**, 310
load-facts ..... **260**, 311
LoadFactsFromString ..... 304
load-instances ..... **284**, 309, 310
local ..... 260
log ..... **186**
log10 ..... **186**
logical name ..... **168**, 255
nil ..... 171, 173
stdin ..... 169, 171, 172, 176, 177, 255
stdout ..... 169, 171, 173, 255
t ..... 169, 171, 172, 173, 176, 177
wclips ..... 169, 255
wdialog ..... 169, 255
wdisplay ..... 169, 255
werror ..... 169, 255
wtrace ..... 169, 255
wwarning ..... 169, 255
logical support ..... **58**, 204, 205, 267
loop-for-count ..... 82, **190**, 192, 247, 314
lowcase ..... **166**
make-instance ..... 8, 47, 82, 92, 94, 96, 101, 108, **115**, 117, 239, 284, 311
matches ..... 63, **264**
math functions ..... **178**, **182**
max ..... **181**
max-number-of-elements ..... 148
member$ ..... **158**, 303, 309
mem-requests ..... **286**
mem-used ..... **285**
message ..... 14, 16, **17**, 18, 75, 87, 94, 103, 105, 112, 113, 114, **115**, 116, 119
dispatch ..... **104**
execution error ..... 105, 114, 237
execution error ..... **114**


Preview from Notesale.co.uk
Page 424 of 428

symbol .....6, **7**, 8, 240, 241
reserved.....12
and .....12
declare.....12
exists .....12
forall .....12
logical .....12
not.....12
object .....12
or.....12
test .....12
symbolp .....**152**
symbol-to-instance-name .....**240**
sym-cat .....**164**
system.....**253**
tab..... 7, 171, 309
tan.....**183**
tanh.....**183**
template .....205
then portion.....**15**
tilde .....7
time .....**196**
timer .....**198**
timer .....304
top level.....**3**
toss .....**292**
trigonometric math functions .....**182**
TrueSymbol .....302
truth maintenance.....**58**
type function..... **223**, 240
unconditional support.....**58**
undefclass .....**276**
undeffacts .....**262**
undeffunction.....**273**
undefgeneric .....**274**
undefglobal ..... 67, **271**
undefinstances .....**282**
undefmessage-handler .....**280**
undefmethod .....**274**
undefrule..... 139, **263**
undeftemplate.....**259**
unmake-instance .....49, **239**
unwatch.....**257**, 310
upcase .....**166**
User's Guide .....v, vii
user-functions.....293
value .....**9**
variable ...5, 7, **9**, 11, 14, 33, 34, 38, 53, 165, **187**
global .....3, 14, 64, **67**, 188, 251
vertical bar .....7
visible .....260
vtab..... 171, 309
watch .....**255**, 257, 313
watch item
activations.....28, **256**
all.....**257**
compilations.....249, **256**
deffunctions.....**256**
facts..... 203, 204, **256**
focus .....**256**
generic-functions.....**256**
globals.....67, **256**
instances.....**257**
message-handlers .....**257**
messages .....**257**
methods.....**256**
rules .....**256**, 268
slots.....**257**
statistics.....**256**, 268
while.....82, **190**, 247, 314
wildcard ..... 33, 34, **36**
wordp.....152

Preview from Notesale.co.uk
Page 428 of 428