Reference Manual

Volume I
Basic Programming Guide

Version 6.40 Beta

November 10th 2017

CLIPS Basic Programming Guide

Version 6.40 Beta November 10th 2017

CONTENTS

License Information

Preface

About CLIPS

Developed at NASA's Johnson Space Center from 1985 to 1996, the 'C' Language Integrated Production System (CLIPS) is a rule-based programming language useful for creating expert systems and other programs where a heuristic solution is easier to implement and maintain than an algorithmic solution. Written in C for portability, CLIPS can be installed and used on a wide variety of platforms. Since 1996, CLIPS has been available as public domain software.

CLIPS Version 6.4

Version 6.4 of CLIPS includes three major enhancements: a redesigned C Application Programming Interface (API); wrapper classes and example programs for .NET and Java; and Integrated Development Environments (IDEs) with Unicode support for Windows and Java. For a detailed listing of differences between releases of CLIPS, refer to appendix B of the *Basic Programming Guide* and appendix B of the *Advanced Programming Guide*.

CLIPS Documentation

Two documents are provided with CLIPS.

- The *CLIPS Reference Manual* which is split into several volumes:
  - *Volume I - The Basic Programming Guide* provides information on the CLIPS programming language.
  - *Volume II - The Advanced Programming Guide* provides information on compiling CLIPS and use of the C Application Programming Interfaces.
  - *Volume III - The Interfaces Guide* provides information on the CLIPS Integrated Development Environments, wrapper classes, and example programs.
- The *CLIPS User's Guide* provides an introduction to CLIPS and rule-based programming.

Section 1:
Introduction

This manual is the *Basic Programming Guide* for CLIPS. It is intended for users interested in the syntax of CLIPS. No previous expert system background is required, although a general understanding of computer languages is assumed. Section 2 of this manual provides an overview of the CLIPS language and basic terminology. Sections 3 through 11 provide additional details regarding the CLIPS programming language on topics such as rules and the CLIPS Object Oriented Programming Language (COOL). The types of actions and functions provided by CLIPS are defined in section 12. Finally, commands typically used from the CLIPS interactive interface are described in section 13.

The *Basic Programming Guide* documents just the basic CLIPS syntax. More advanced capabilities, such as user-defined functions, embedded applications, etc., are documented more fully in the *Advanced Programming Guide*. The *Advanced Programming Guide* is intended for users who have a complete knowledge of the CLIPS syntax and a programming background. It is *not* necessary to read the *Advanced Programming Guide* to learn how to use CLIPS. CLIPS can be learned and simple expert systems can be built with the information provided in this manual.

Section 2:
CLIPS Overview

This section gives a general overview of CLIPS and of the basic concepts used throughout this manual.

2.1 Interacting with CLIPS

CLIPS expert systems may be executed in three ways: interactively using a simple, text-oriented, command prompt interface; interactively using a window/menu/mouse interface on certain machines; or as embedded expert systems in which the user provides a main program and controls execution of the expert system. Embedded applications are discussed in the *Advanced Programming Guide*. In addition, a series of commands can be automatically read directly from a file when CLIPS is first started or as the result of the **batch** command.

The generic CLIPS interface is a simple, interactive, text-oriented, command prompt interface for high portability. The standard usage is to create or edit a knowledge base using any standard text editor, save the knowledge base as one or more text files, exit the editor and execute CLIPS, then load the knowledge base into CLIPS. The interface provides commands for viewing the current state of the system, tracing execution, adding or removing information, and clearing CLIPS.

A more sophisticated window interface is available for the Macintosh, Windows, and Java environments. All interface commands described in this section are available in the window interfaces. These interfaces are described in more detail in the *Interfaces Guide*.

2.1.1 Top-Level Commands

The primary method for interacting with CLIPS in a non-embedded environment is through the CLIPS **command prompt** (or **top-level**). When the "CLIPS>" prompt is printed, a command may be entered for evaluation. Commands may be function calls, constructs, local or global variables, or constants. If a function call is entered (see section 2.3.2), that function is evaluated and its return value is printed. Function calls in CLIPS use a prefix notation—the operands to a function always appear after the function name. Entering a construct definition (see section 2.3.3) at the CLIPS prompt creates a new construct of the appropriate type. Entering a global variable (see section 2.4.3) causes the value of the global variable to be printed. Local variables can be set at the command prompt using the bind function and retain their value until a reset or clear command is issued. Entering a local variable at the command prompt causes the value of

the local variable to be printed. Entering a constant (see section 2.3.1) at the top-level causes the constant to be printed (which is not very useful). For example,

```
         CLIPS (V6.40 3/08/17)
CLIPS> (+ 3 4)
7
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. The constant symbol *red* was entered and was returned (since a constant evaluates to itself). A local variable ?a is assigned the value 5 using the bind function. The addition function is called to add the variable ?a to the integer 3 yielding 8. The reset command is called to reset the CLIPS environment (which among other effects removes the assignment of local variables). When the variable ?a is entered at the prompt, an error occurs because the variable is no longer bound.

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:

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.

For the **-l** option, <filename> should be a file containing CLIPS constructs. This file will be loaded into the environment. The **-l** command line option is equivalent to interactively entering a **load** command.

Files specified using the **–f** option are not processed until the command prompt appears, so these files will always be processed after files specified using the **–f2** and **–l** options.

2.1.3 Integration with Other Programming Languages

When using an expert system, two kinds of integration are important: embedding CLIPS in other systems, and calling external functions from CLIPS. CLIPS was designed to allow both kinds of integration.

Using CLIPS as an embedded application allows the easy integration of CLIPS with existing systems. This is useful in cases where the expert system is a small part of a larger task or needs to share data with other functions. In these situations, CLIPS can be called as a subroutine and information may be passed to and from CLIPS. Embedded applications are discussed in the *Advanced Programming Guide*.

It also may be useful to call external functions while executing a CLIPS construct or from the top-level of the interactive interface. CLIPS variables or literal values may be passed to an external function, and functions may return values to CLIPS. The easy addition of external functions allows CLIPS to be extended or customized in almost any way. The *Advanced Programming Guide* describes how to integrate CLIPS with functions or systems written in C as well as in other languages.

2.2 Reference Manual Syntax

The terminology used throughout this manual to describe the CLIPS syntax is fairly common to computer reference manuals. Plain words or characters, particularly parentheses, are to be typed exactly as they appear. Bolded words or characters, however, represent a verbal description of what is to be entered. Sequences of words enclosed in single-angle brackets (called terms or non-terminal symbols), such as <string>, represent a single entity of the named class of items to

be supplied by the user. A non-terminal symbol followed by a *, represents *zero or more* entities of the named class of items which must be supplied by the user. A non-terminal symbol followed by a +, represents *one or more* entities of the named class of items which must be supplied by the user. A * or + by itself is to be typed as it appears. Vertical and horizontal ellipsis (three dots arranged respectively vertically and horizontally) are also used between non-terminal symbols to indicate the occurrence of one or more entities. A term enclosed within square brackets, such as [<comment>], is optional (i.e. it may or may not be included). Vertical bars indicate a choice between multiple terms. White spaces (tabs, spaces, carriage returns) are used by CLIPS only as delimiters between terms and are ignored otherwise (unless inside double quotes). The ::= symbol is used to indicate how a non-terminal symbol can be replaced. For example, the following syntax description indicates that a <lexeme> can be replaced with either a <symbol> or a <string>.

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

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

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.

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          B76-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 *Advanced Programming Guide*. Within CLIPS, the printed representation of an external-address is

```
<Pointer-XXXXXX>
```

where XXXXXX is the external-address.

A **fact** is a list of atomic values that are either referenced positionally (ordered facts) or by name (non-ordered or template facts). Facts are referred to by index or address; section 2.4.1 gives more details. The printed format of a **fact-address** is:

```
<Fact-XXX>
```

where XXX is the fact-index.

An **instance** is an **object** that is an instantiation or specific example of a **class**. Objects in CLIPS are defined to be floats, integers, symbols, strings, multifield values, external-addresses, fact-addresses or instances of a user-defined class. A user-defined class is created using the **defclass** construct. An instance of a user-defined class is created with the **make-instance** function, and such an instance can be referred to uniquely by address. Within the scope of a module (see section 10.5.1), an instance can also be uniquely referred to by name. All of these definitions will be covered in more detail in Sections 2.4.2, 2.5.2.3, 2.6 and 9. An **instance-name** is formed by enclosing a symbol within left and right brackets. Thus, pure symbols may not be surrounded by brackets. If the CLIPS Object Oriented Language (COOL) is not included in a particular CLIPS configuration, then brackets may be wrapped around symbols. Some examples of instance-names are:

```
[pump-1]      [foo]          [+++]          [123-890]
```

Note that the brackets are not part of the name of the instance; they merely indicate that the enclosed symbol is an instance-name. An **instance-address** can only be obtained by binding the return value of a function called **instance-address** or by binding a variable to an instance matching an object pattern on the LHS of a rule (i.e., it is not possible to specify an instance-address by typing the value). A reference to an instance of a user-defined class can

either be by name or address; instance-addresses should only be used when speed is critical. Within CLIPS, the printed representation of an instance-address is

```
<Instance-XXX>
```

where XXX is the name of the instance.

In CLIPS, a placeholder that has a value (one of the primitive data types) is referred to as a **field**. The primitive data types are referred to as **single-field values**. A **constant** is a non-varying single field value directly expressed as a series of characters (which means that external-addresses, fact-addresses and instance-addresses cannot be expressed as constants because they can only be obtained through function calls and variable bindings). A **multifield value** is a sequence of zero or more single field values. When displayed by CLIPS, multifield values are enclosed in parentheses. Collectively, single and multifield values are referred to as **values**. Some examples of multifield values are

```
(a)          (1 bar foo)    ()             (x 3.0 "red" 567)
```

Note that the multifield value (a) is not the same as the single field value *a*. Multifield values are created either by calling functions which return multifield values, by using wildcard arguments in a deffunction, object message-handler, or method, or by binding variables during the pattern-matching process for rules. In CLIPS, a **variable** is a symbolic location that is used to store values. Variables are used by many of the CLIPS constructs (such as defrule, deffunction, defmethod, and defmessage-handler) and their usage is explained in the sections describing each of these constructs.

2.3.2 Functions

A **function** in CLIPS is a piece of executable code identified by a specific name which returns a useful value or performs a useful side effect (such as displaying information). Throughout the CLIPS documentation, the word function is generally used to refer only to functions which return a value (whereas commands and actions are used to refer to functions which have a side effect but generally do not return a value).

There are several types of functions. **User defined functions** and **system defined functions** are pieces of code that have been written in an external language (such as C, FORTRAN, or Ada) and linked with the CLIPS environment. System defined functions are those functions that have been defined internally by the CLIPS environment. User defined functions are functions that have been defined externally of the CLIPS environment. A complete list of system defined functions can be found in appendix H.

The **deffunction** construct allows users to define new functions directly in the CLIPS environment using CLIPS syntax. Functions defined in this manner appear and act like other

functions, however, instead of being directly executed (as code written in an external language would be) they are interpreted by the CLIPS environment. Deffunctions are also discussed in section 2.5.2.1 in the context of procedural knowledge representation.

Generic functions can be defined using the **defgeneric** and **defmethod** constructs. Generic functions allow different pieces of code to be executed depending upon the arguments passed to the generic function. Thus, a single function name can be **overloaded** with more than one piece of code. Generic functions are also discussed in section 2.5.2.2 in the context of procedural knowledge representation.

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.

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.

2.4.1 Facts

Facts are one of the basic high-level forms for representing information in a CLIPS system. Each **fact** represents a piece of information that has been placed in the current list of facts, called the **fact-list**. Facts are the fundamental unit of data used by rules (see section 2.5.1).

Facts may be added to the fact-list (using the **assert** command), removed from the fact-list (using the **retract** command), modified (using the **modify** command), or duplicated (using the **duplicate** command) through explicit user interaction or as a CLIPS program executes. The number of facts in the fact-list and the amount of information that can be stored in a fact is limited only by the amount of available memory. If a fact is asserted into the fact-list that exactly matches an already existing fact, the new assertion will be ignored (however, this behavior can be changed, see sections 13.4.4 and 13.4.5).

Some commands, such as the **retract**, **modify**, and **duplicate** commands, require a fact to be specified. A fact can be specified either by **fact-index** or **fact-address**. Whenever a fact is asserted it is given a unique integer index called a fact-index. Fact-indices start at one and are incremented by one for each new fact. When a fact is modified, its fact-index remains unchanged. Whenever a **reset** or **clear** command is given, the fact-indices restart at one. A fact may also be specified through the use of a fact-address. A fact-address can be obtained by capturing the return value of commands which return fact addresses (such as **assert**, **modify**, and **duplicate**) or by binding a variable to the fact address of a fact which matches a pattern on the LHS of a rule (see section 5.4.1.8 for details).

A **fact identifier** is a shorthand notation for displaying a fact. It consists of the character “f”, followed by a dash, followed by the fact-index of the fact. For example, f-10 refers to the fact with fact-index 10.

A fact is stored in one of two formats: ordered or non-ordered.

2.4.1.1 Ordered Facts

Ordered facts consist of a symbol followed by a sequence of zero or more fields separated by spaces and delimited by an opening parenthesis on the left and a closing parenthesis on the right. The first field of an ordered fact specifies a "relation" that applied to the remaining fields in the ordered fact. For example, (father-of jack bill) states that bill is the father of jack.

Some examples of ordered facts are shown following.

```
(the pump is on)
(altitude is 10000 feet)
(grocery-list bread milk eggs)
```

Fields in a non-ordered fact may be of any of the primitive data types (with the exception of the first field which must be a symbol), and no restriction is placed on the ordering of fields. The following symbols are reserved and should not be used as the *first* field in any fact (ordered or non-ordered): *test*, *and*, *or*, *not*, *declare*, *logical*, *object*, exists, and forall. These words are reserved only when used as a deftemplate name (whether explicitly defined or implied). These symbols may be used as slot names, however, this is not recommended.

2.4.1.2 Non-ordered Facts

Ordered facts encode information positionally. To access that information, a user must know not only what data is stored in a fact but which field contains the data. **Non-ordered (or deftemplate) facts** provide the user with the ability to abstract the structure of a fact by assigning names to each field in the fact. The **deftemplate** construct (see section 3) is used to create a template that can then be used to access fields by name. The deftemplate construct is analogous to a record or structure definition in programming languages such as Pascal and C.

The deftemplate construct allows the name of a template to be defined along with zero or more definitions of **named fields** or **slots**. Unlike ordered facts, the slots of a deftemplate fact may be constrained by type, value, and numeric range. In addition, default values can be specified for a slot. A slot consists of an opening parenthesis followed by the name of the slot, zero or more fields, and a closing parenthesis. Note that slots may not be used in an ordered fact and that positional fields may not be used in a deftemplate fact.

Deftemplate facts are distinguished from ordered facts by the first field within the fact. The first field of all facts must be a symbol, however, if that symbol corresponds to the name of a deftemplate, then the fact is a deftemplate fact. The first field of a deftemplate fact is followed by a list of zero or more slots. As with ordered facts, deftemplate facts are enclosed by an opening parenthesis on the left and a closing parenthesis on the right.

Some examples of deftemplate facts are shown following.

```
(client (name "Joe Brown") (id X9345A))
(point-mass (x-velocity 100) (y-velocity -200))
(class (teacher "Martha Jones") (#-students 30) (Room "37A"))
(grocery-list (#-of-items 3) (items bread milk eggs))
```

Note that the order of slots in a deftemplate fact is not important. For example the following facts are all identical:

```
(class (teacher "Martha Jones") (#-students 30) (Room "37A"))
(class (#-students 30) (teacher "Martha Jones") (Room "37A"))
(class (Room "37A") (#-students 30) (teacher "Martha Jones"))
```

In contrast, note that the following ordered fact *are not* identical.

```
(class "Martha Jones" 30 "37A")
(class 30 "Martha Jones" "37A")
(class "37A" 30 "Martha Jones")
```

The immediate advantages of clarity and slot order independence for deftemplate facts should be readily apparent.

In addition to being asserted and retracted, deftemplate facts can also be modified and duplicated (using the **modify** and **duplicate** commands). Modifying a fact changes a set of specified slots within that fact. Duplicating a fact creates a new fact identical to the original fact and then changes a set of specified slots within the new fact. The benefit of using the modify and duplicate commands is that slots which don't change, don't have to be specified.

2.4.1.3 Initial Facts

The **deffacts** construct allows a set of *a priori* or initial knowledge to be specified as a collection of facts. When the CLIPS environment is reset (using the **reset** command) every fact specified within a deffacts construct in the CLIPS knowledge base is added to the fact-list.

2.4.2 Objects

An **object** in CLIPS is defined to be a symbol, a string, a floating-point or integer number, a multifield value, an external-address or an instance of a user-defined class. Section 2.3.1 explains how to reference instances of user-defined classes. Objects are described in two basic parts: properties and behavior. A **class** is a template for common properties and behavior of objects that are **instances** of that class. Some examples of objects and their classes are:

Object (Printed Representation)	Class
Rolls-Royce	SYMBOL
"Rolls-Royce"	STRING
8.0	FLOAT
8	INTEGER
(8.0 Rolls-Royce 8 [Rolls-Royce])	MULTIFIELD
<Pointer-00CF61AB>	EXTERNAL-ADDRESS
[Rolls-Royce]	CAR (a user-defined class)

Objects in CLIPS are split into two important categories: primitive types and instances of *user-defined* classes. These two types of objects differ in the way they are referenced, created and deleted as well as how their properties are specified.

Primitive type objects are referenced simply by giving their value, and they are created and deleted implicitly by CLIPS as they are needed. Primitive type objects have no names or slots, and their classes are predefined by CLIPS. The behavior of primitive type objects is like that of instances of user-defined classes, however, in that you can define message-handlers and attach them to the primitive type classes. It is anticipated that primitive types will not be used often in an object-oriented programming (OOP) context; the main reason classes are provided for them is for use in generic functions. Generic functions use the classes of their arguments to determine which methods to execute; sections 2.3.2, 2.5.2.2 and 8 give more detail.

An instance of a user-defined class is referenced by name or address, and they are created and deleted explicitly via messages and special functions. The properties of an instance of a *user-defined* class are expressed by a set of slots, which the object obtains from its class. As previously defined, slots are named single field or multifield values. For example, the object Rolls-Royce is an instance of the class CAR. One of the slots in class CAR might be “price”, and the Rolls-Royce object’s value for this slot might be $75,000.00. The behavior of an object is specified in terms of procedural code called message-handlers, which are attached to the object’s class. Message-handlers and manipulation of objects are described in Section 2.5.2.3. All instances of a user-defined class have the same set of slots, but each instance may have different values for those slots. However, two instances that have the same set of slots do not necessarily belong to the same class, since two different classes can have identical sets of slots.

The primary difference between object slots and template (or non-ordered) facts is the notion of inheritance. Inheritance allows the properties and behavior of a class to be described in terms of other classes. COOL supports multiple inheritance: a class may directly inherit slots and message-handlers from more than one class. Since inheritance is only useful for slots and message-handlers, it is often not meaningful to inherit from one of the primitive type classes,

such as MULTIFIELD or NUMBER. This is because these classes cannot have slots and usually do not have message-handlers.

Further discussion on these topics can be found in Section 2.6, and a comprehensive description of the CLIPS Object-Oriented Language (COOL) can be found in Section 9.

2.4.2.1 Initial Objects

The **definstances** construct allows a set of *a priori* or initial knowledge to be specified as a collection of instances of user-defined classes. When the CLIPS environment is reset (using the **reset** command) every instance specified within a definstances construct in the CLIPS knowledge base is added to the instance-list.

2.4.3 Global Variables

The **defglobal** construct allows variables to be defined which are global in scope throughout the CLIPS environment. That is, a global variable can be accessed anywhere in the CLIPS environment and retains its value independent of other constructs. In contrast, some constructs (such as defrule and deffunction) allow local variables to be defined within the definition of the construct. These local variables can be referred to within the construct, but have no meaning outside the construct. A CLIPS global variable is similar to global variables found in procedural programming languages such as LISP, C and Ada. Unlike C and Ada, however, CLIPS global variables are weakly typed (they are not restricted to holding a value of a single data type).

2.5 Knowledge Representation

CLIPS provides heuristic and procedural paradigms for representing knowledge. These two paradigms are discussed in this section. Object-oriented programming (which combines aspects of both data abstraction and procedural knowledge) is discussed in section 2.6.

2.5.1 Heuristic Knowledge – Rules

One of the primary methods of representing knowledge in CLIPS is a rule. Rules are used to represent heuristics, or “rules of thumb”, which specify a set of actions to be performed for a given situation. The developer of an expert system defines a set of rules that collectively work together to solve a problem. A **rule** is composed of an **antecedent** and a **consequent**. The antecedent of a rule is also referred to as the **if portion** or the **left-hand side** (LHS) of the rule. The consequent of a rule is also referred to as the **then portion** or the **right-hand side** (RHS) of the rule.

The antecedent of a rule is a set of **conditions** (or **conditional elements**) that must be satisfied for the rule to be applicable. In CLIPS, the conditions of a rule are satisfied based on the existence or non-existence of specified facts in the fact-list or specified instances of user-defined classes in the instance-list. One type of condition that can be specified is a **pattern**. Patterns consist of a set of restrictions that are used to determine which facts or objects satisfy the condition specified by the pattern. The process of matching facts and objects to patterns is called **pattern-matching**. CLIPS provides a mechanism, called the **inference engine**, which automatically matches patterns against the current state of the fact-list and instance-list and determines which rules are applicable.

The consequent of a rule is the set of actions to be executed when the rule is applicable. The actions of applicable rules are executed when the CLIPS inference engine is instructed to begin execution of applicable rules. If more than one rule is applicable, the inference engine uses a **conflict resolution strategy** to select which rule should have its actions executed. The actions of the selected rule are executed (which may affect the list of applicable rules) and then the inference engine selects another rule and executes its actions. This process continues until no applicable rules remain.

In many ways, rules can be thought of as IF-THEN statements found in procedural programming languages such as C and Ada. However, the conditions of an IF-THEN statement in a procedural language are only evaluated when the program flow of control is directly at the IF-THEN statement. In contrast, rules act like WHENEVER-THEN statements. The inference engine always keeps track of rules that 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.

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.

2.5.2.3 Object Message-Passing

Objects are described in two basic parts: properties and behavior. Object properties are specified in terms of slots obtained from the object's class; slots are discussed in more detail in Section 2.4.2. Object behavior is specified in terms of procedural code called message-handlers which are attached to the object's class. Objects are manipulated via message-passing. For example, to cause the Rolls-Royce object, which is an instance of the class CAR, to start its engine, the user must call the **send** function to send the message "start-engine" to the Rolls-Royce. How the Rolls-Royce responds to this message will be dictated by the execution of the message-handlers for "start-engine" attached to the CAR class and any of its superclasses. The result of a message is similar to a function call in CLIPS: a useful return value or side-effect.

Further discussion on message-handlers can be found in Section 2.6, and a comprehensive description of the CLIPS Object-Oriented Language (COOL) can be found in Section 9.

2.5.2.4 Defmodules

Defmodules allow a knowledge base to be partitioned. Every construct defined must be placed in a module. The programmer can explicitly control which constructs in a module are visible to other modules and which constructs from other modules are visible to a module. The visibility of facts and instances between modules can be controlled in a similar manner. Modules can also be

used to control the flow of execution of rules. Defmodules are covered comprehensively in Section 10.

2.6 CLIPS Object-Oriented Language

This section gives a brief overview of the programming elements of the CLIPS Object-Oriented Language (COOL). COOL includes elements of data abstraction and knowledge representation. This section gives an overview of COOL as a whole, incorporating the elements of both concepts. References to instanes of user-defined classes are discussed in Section 2.3.1, and the structure of objects is discussed in Sections 2.4.2 and 2.5.2.3. The comprehensive details of COOL are given in Section 9.

2.6.1 COOL Deviations from a Pure OOP Paradigm

In a pure OOP language, *all* programming elements are objects which can only be manipulated via messages. In CLIPS, the definition of an object is much more constrained: floating-point and integer numbers, symbols, strings, multifield values, external-addresses, fact-addresses and instances of user-defined classes. All objects *may* be manipulated with messages, except instances of user-defined classes, which *must* be. For example, in a pure OOP system, to add two numbers together, you would send the message "add" to the first number object with the second number object as an argument. In CLIPS, you may simply call the "+" function with the two 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 that 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.

The definition 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 that 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 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.

2.6.3 Instance-set Queries and Distributed Actions

In addition to the ability of rules to directly pattern-match on objects, COOL provides a useful query system for determining, grouping and performing actions on sets of instances of user-defined classes that meet user-defined criteria. The query system allows you to associate instances that are either related or not. You can simply use the query system to determine if a particular association set exists, you can save the set for future reference, or you can iterate an action over the set. An example of the use of the query system might be to find the set of all pairs of boys and girls that have the same age.

Section 3:
Deftemplate Construct

Ordered facts encode information positionally. To access that information, a user must know not only what data is stored in a fact but also which field contains the data. Non-ordered (or deftemplate) facts provide the user with the ability to abstract the structure of a fact by assigning names to each field found within the fact. The **deftemplate** construct is used to create a template that can then be used by non-ordered facts to access fields of the fact by name. The deftemplate construct is analogous to a record or structure definition in programming languages such as Pascal and C.

Syntax

```
(deftemplate <deftemplate-name> [<comment>]
   <slot-definition>*)

<slot-definition> ::= <single-slot-definition> |
                      <multislot-definition>

<single-slot-definition>
               ::= (slot <slot-name>
                         <template-attribute>*)

<multislot-definition>
               ::= (multislot <slot-name>
                              <template-attribute>*)

<template-attribute> ::= <default-attribute> |
                         <constraint-attribute>

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

Redefining a deftemplate will result in the previous definition being discarded. A deftemplate can not be redefined while it is being used (for example, by a fact or pattern in a rule). A deftemplate can have any number of single or multifield slots. CLIPS always enforces the single and multifield definitions of the deftemplate. For example, it is an error to store (or match) multiple values in a single-field slot.

Example

```
(deftemplate thing
   (slot name)
   (slot location)
   (slot on-top-of)
   (slot weight)
   (multislot contents))
```

3.1 Slot Default Values

The <default-attribute> specifies the value to be used for unspecified slots of a template fact when an **assert** action is performed. One of two types of default selections can be chosen: default or dynamic-default.

The **default** attribute specifies a static default value. The specified expressions are evaluated once when the deftemplate is defined and the result is stored with the deftemplate. The result is assigned to the appropriate slot when a new template fact is asserted. If the keyword ?DERIVE is used for the default value, then a default value is derived from the constraints for the slot (see section 11.5 for more details). By default, the default attribute for a slot is (default ?DERIVE). If the keyword ?NONE is used for the default value, then a value must explicitly be assigned for a slot when an assert is performed. It is an error to assert a template fact without specifying the values for the (default ?NONE) slots.

The **default-dynamic** attribute is a dynamic default. The specified expressions are evaluated every time a template fact is asserted, and the result is assigned to the appropriate slot.

A single-field slot may only have a single value for its default. Any number of values may be specified as the default for a multifield slot (as long as the number of values satisfies the cardinality attribute for the slot).

Example

```
CLIPS> (clear)
CLIPS>
(deftemplate point
   (slot x (default ?NONE))
   (slot y (type INTEGER) (default ?DERIVE))
   (slot id (default (gensym*)))
   (slot uid (default-dynamic (gensym*))))
CLIPS> (assert (point))

[TMPLTRHS1] Slot x requires a value because of its (default ?NONE) attribute.
CLIPS> (assert (point (x 3)))
<Fact-1>
CLIPS> (assert (point (x 4)))
<Fact-2>
CLIPS> (facts)
f-1     (point (x 3) (y 0) (id gen1) (uid gen2))
f-2     (point (x 4) (y 0) (id gen1) (uid gen3))
For a total of 2 facts.
CLIPS>
```

3.2 Slot Default Constraints for Pattern-Matching

Single-field slots that are not specified in a pattern on the LHS of a rule are defaulted to single-field wildcards (?) and multifield slots are defaulted to multifield wildcards ($?).

3.3 Slot Value Constraint Attributes

The syntax and functionality of single and multifield constraint attributes are described in detail in Section 11. Static and dynamic constraint checking for deftemplates is supported. Static checking is performed when constructs or commands using deftemplates slots are being parsed (and the specific deftemplate associated with the construct or command can be immediately determined). Template patterns used on the LHS of a rule are also checked to determine if constraint conflicts exist among variables used in more than one slot. Errors for inappropriate values are immediately signaled. References to fact-indexes made in commands such as **modify** and **duplicate** are considered to be ambiguous and are never checked using static checking. Static checking is always enabled. Dynamic checking is also supported. If dynamic checking is enabled, then new deftemplate facts have their values checked when added to the fact-list. This dynamic checking is disabled by default. This behavior can be changed using the **set-dynamic-constraint-checking** function. If a violation occurs when dynamic checking is being performed, then execution will be halted.

Example

```
(deftemplate thing
   (slot name
      (type SYMBOL)
      (default ?DERIVE))
   (slot location
      (type SYMBOL)
      (default ?DERIVE))
   (slot on-top-of
      (type SYMBOL)
      (default floor))
   (slot weight
      (allowed-values light heavy)
      (default light))
   (multislot contents
      (type SYMBOL)
      (default ?DERIVE)))
```

3.4 Implied Deftemplates

Asserting or referring to an ordered fact (such as in a LHS pattern) creates an “implied” deftemplate with a single implied multifield slot. The implied multifield slot’s name is not printed when the fact is printed. The implied deftemplate can be manipulated and examined identically to any user defined deftemplate (although it has no pretty print form).

Example

```
CLIPS> (clear)
CLIPS> (assert (groceries milk eggs cheese))
<Fact-1>
CLIPS> (defrule study (homework math) =>)
CLIPS> (list-deftemplates)
groceries
homework
```

```
For a total of 2 deftemplates.
CLIPS> (facts)
f-1     (groceries milk eggs cheese)
For a total of 1 fact.
CLIPS>
```

Section 4:
Deffacts Construct

With the **deffacts** construct, a list of facts can be defined which are automatically asserted whenever the **reset** command is performed. Facts asserted through deffacts may be retracted or pattern-matched like any other fact. The initial fact-list, including any defined deffacts, is always reconstructed after a **reset** command.

Syntax

```
(deffacts <deffacts-name> [<comment>]
   <RHS-pattern>*)
```

Redefining a currently existing deffacts causes the previous deffacts with the same name to be removed even if the new definition has errors in it. There may be multiple deffacts constructs and any number of facts (either ordered or deftemplate) may be asserted into the initial fact-list by each deffacts construct.

Dynamic expressions may be included in a fact by embedding the expression directly within the fact. All such expressions are evaluated when CLIPS is reset.

Example

```
CLIPS> (clear)
CLIPS>
(deftemplate oav
   (slot object)
   (slot attribute)
   (slot value))
CLIPS>
(deffacts startup "Refrigerator Status"
   (oav (object refrigerator)
        (attribute light)
        (value on))
   (oav (object refrigerator)
        (attribute door)
        (value open))
   (oav (object refrigerator)
        (attribute temp)
        (value 40)))
CLIPS> (facts)
CLIPS> (reset)
CLIPS> (facts)
f-1     (oav (object refrigerator) (attribute light) (value on))
f-2     (oav (object refrigerator) (attribute door) (value open))
f-3     (oav (object refrigerator) (attribute temp) (value 40))
For a total of 3 facts.
CLIPS>
```

Section 5:
Defrule Construct

One of the primary methods of representing knowledge in CLIPS is a rule. A **rule** is a collection of conditions and the actions to be taken if the conditions are met. The developer of an expert system defines the rules that describe how to solve a problem. Rules execute (or **fire**) based on the existence or non-existence of facts or instances of user-defined classes. CLIPS provides the mechanism (the **inference engine**) which attempts to match the rules to the current state of the system (as represented by the fact-list and instance-list) and applies the actions.

Throughout this section, the term **pattern entity** will be used to refer to either a fact or an instance of a user-defined class.

5.1 Defining Rules

Rules are defined using the **defrule** construct.

Syntax

```
(defrule <rule-name> [<comment>]
   [<declaration>]               ; Rule Properties
   <conditional-element>*        ; Left-Hand Side (LHS)
   =>
   <action>*)                    ; Right-Hand Side (RHS)
```

Redefining a currently existing defrule causes the previous defrule with the same name to be removed even if the new definition has errors in it. The LHS is made up of a series of conditional elements (CEs) that typically consist of pattern conditional elements (or just simply patterns) to be matched against pattern entities. An implicit **and** conditional element always surrounds all the patterns on the LHS. The RHS contains a list of actions to be performed when the LHS of the rule is satisfied. In addition, the LHS of a rule may also contain declarations about the rule's properties immediately following the rule's name and comment (see section 5.4.10 for more details). The arrow (**=>**) separates the LHS from the RHS. There is no limit to the number of conditional elements or actions a rule may have (other than the limitation placed by actual available memory). Actions are performed sequentially if, and only if, all conditional elements on the LHS are satisfied.

If no conditional elements are on the LHS, the rule will automatically be activated. If no actions are on the RHS, the rule can be activated and fired but nothing will happen.

As rules are defined, they are incrementally reset. This means that CEs in newly defined rules can be satisfied by pattern entities at the time the rule is defined, in addition to pattern entities created after the rule is defined (see section 13.1.8).

Example

```
CLIPS> (clear)
CLIPS>
(deftemplate oav
   (slot object)
   (slot attribute)
   (slot value))
CLIPS>
(defrule example-rule   "This is an example of a simple rule"
   (oav (object refrigerator)
        (attribute light)
        (value on))
   (oav (object refrigerator)
        (attribute door)
        (value open))
   =>
   (assert (oav (object refrigerator)
                (attribute food)
                (value spoiled))))
CLIPS>
(assert (oav (object refrigerator)
             (attribute light)
             (value on))
        (oav (object refrigerator)
             (attribute door)
             (value open)))
<Fact-2>
CLIPS> (agenda)
0      example-rule: f-1,f-2
For a total of 1 activation.
CLIPS> (run)
CLIPS> (facts)
f-1     (oav (object refrigerator) (attribute light) (value on))
f-2     (oav (object refrigerator) (attribute door) (value open))
f-3     (oav (object refrigerator) (attribute food) (value spoiled))
For a total of 3 facts.
CLIPS>
```

5.2 Basic Cycle Of Rule Execution

Once a knowledge base (in the form of rules) is built and the fact-list and instance-list is prepared, CLIPS is ready to execute rules. In a conventional language the programmer explicitly defines the starting point, the stopping point, and the sequence of operations. With CLIPS, the program flow does not need to be defined quite so explicitly. The knowledge (rules) and the data (facts and instances) are separated, and the inference engine provided by CLIPS is used to apply the knowledge to the data. The basic execution cycle is as follows:

a) If the rule firing limit has been reached or there is no current focus, then execution is halted. Otherwise, the top rule on the agenda of the module that is the current focus is selected for

execution. If there are no rules on that agenda, then the current focus is removed from the focus stack and the current focus becomes the next module on the focus stack. If the focus stack is empty, then execution is halted, otherwise step *a* is executed again. See sections 5.4.10.2, 10.6, 12.12, and 13.7 for information on the focus stack and the current focus.

b) The right-hand side (RHS) actions of the selected rule are executed. The use of the **return** function on the RHS of a rule may remove the current focus from the focus stack (see sections 10.6 and 12.6.7). The number of rules fired is incremented for use with the rule firing limit.

c) As a result of step b, rules may be **activated** or **deactivated**. Activated rules (those rules whose conditions are currently satisfied) are placed on the **agenda** of the module in which they are defined. The placement on the agenda is determined by the **salience** of the rule and the current **conflict resolution strategy** (see sections 5.3, 5.4.10, 13.7.5, and 13.7.6). Deactivated rules are removed from the agenda. If the activations item is being watched (see section 13.2.3), then an informational message will be displayed each time a rule is activated or deactivated.

d) If **dynamic salience** is being used, the salience values for all rules on the agenda are reevaluated (see sections 5.4.10, 13.7.9, and 13.7.10). Repeat the cycle beginning with step a.

5.3 Conflict Resolution Strategies

The **agenda** is the list of all rules that have their conditions satisfied (and have not yet been executed). Each module has its own agenda. The agenda acts similar to a stack (the top rule on the agenda is the first one to be executed). When a rule is newly activated, its placement on the agenda is based (in order) on the following factors:

a) Newly activated rules are placed above all rules of lower salience and below all rules of higher salience.

b) Among rules of equal salience, the current conflict resolution strategy is used to determine the placement among the other rules of equal salience.

c) If a rule is activated (along with several other rules) by the same assertion or retraction of a fact, and steps a and b are unable to specify an ordering, then the rule is arbitrarily (*not randomly*) ordered in relation to the other rules with which it was activated. Note, in this respect, the order in which rules are defined has an arbitrary effect on conflict resolution (which is highly dependent upon the current underlying implementation of rules). *Do not* depend upon this arbitrary ordering for the proper execution of your rules.

CLIPS provides seven conflict resolution strategies: depth, breadth, simplicity, complexity, lex, mea, and random. The default strategy is depth. The current strategy can be set by using the **set-strategy** command (which will reorder the agenda based upon the new strategy).

5.3.1 Depth Strategy

Newly activated rules are placed above all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-3 and rule-4 will be above rule-1 and rule-2 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.

5.3.2 Breadth Strategy

Newly activated rules are placed below all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-1 and rule-2 will be above rule-3 and rule-4 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.

5.3.3 Simplicity Strategy

Among rules of the same salience, newly activated rules are placed above all activations of rules with equal or higher specificity. The **specificity** of a rule is determined by the number of comparisons that must be performed on the LHS of the rule. Each comparison to a constant or previously bound variable adds one to the specificity. Each function call made on the LHS of a rule as part of the :, =, or test conditional element adds one to the specificity. The boolean functions **and**, **or**, and **not** do not add to the specificity of a rule, but their arguments do. Function calls made within a function call do not add to the specificity of a rule. For example, the following rule

```
(deftemplate point
   (slot x)
   (slot y)
   (slot z))

(defrule example
   (point (x ?x) (y ?y) (z ?x))
   (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
   =>)
```

has a specificity of 5. The comparison to the constant item, the comparison of ?x to its previous binding, and the calls to the **numberp**, <, and > functions each add one to the specificity for a total of 5. The calls to the **and** and + functions do not add to the specificity of the rule.

5.3.4 Complexity Strategy

Among rules of the same salience, newly activated rules are placed above all activations of rules with equal or lower specificity.

5.3.5 LEX Strategy

Among rules of the same salience, newly activated rules are placed using the OPS5 strategy of the same name. First the recency of the pattern entities that activated the rule is used to determine where to place the activation. Every fact and instance is marked internally with a "time tag" to indicate its relative recency with respect to every other fact and instance in the system. The pattern entities associated with each rule activation are sorted in descending order for determining placement. An activation with a more recent pattern entities is placed before activations with less recent pattern entities. To determine the placement order of two activations, compare the sorted time tags of the two activations one by one starting with the largest time tags. The comparison should continue until one activation's time tag is greater than the other activation's corresponding time tag. The activation with the greater time tag is placed before the other activation on the agenda.

If one activation has more pattern entities than the other activation and the compared time tags are all identical, then the activation with more time tags is placed before the other activation on the agenda. If two activations have the exact same recency, the activation with the higher specificity is placed above the activation with the lower specificity. Unlike OPS5, the *not* conditional elements in CLIPS have pseudo time tags that are used by the LEX conflict resolution strategy. The time tag of a *not* CE is always less than the time tag of a pattern entity, but greater than the time tag of a *not* CE that was instantiated after the *not* CE in question.

As an example, the following six activations have been listed in their LEX ordering (where the comma at the end of the activation indicates the presence of a *not* CE). Note that a fact's time tag is not necessarily the same as it's index (since instances are also assigned time tags), but if one fact's index is greater than another facts's index, then it's time tag is also greater. For this example, assume that the time tags and indices are the same.

```
rule-6: f-1,f-4
rule-5: f-1,f-2,f-3,
rule-1: f-1,f-2,f-3
rule-2: f-3,f-1
rule-4: f-1,f-2,
rule-3: f-2,f-1
```

Shown following are the same activations with the fact indices sorted as they would be by the LEX strategy for comparison.

```
rule-6: f-4,f-1
rule-5: f-3,f-2,f-1,
rule-1: f-3,f-2,f-1
```

```
rule-2: f-3,f-1
rule-4: f-2,f-1,
rule-3: f-2,f-1
```

5.3.6 MEA Strategy

Among rules of the same salience, newly activated rules are placed using the OPS5 strategy of the same name. First the time tag of the pattern entity associated with the first pattern is used to determine where to place the activation. An activation thats first pattern's time tag is greater than another activations first pattern's time tag is placed before the other activation on the agenda. If both activations have the same time tag associated with the first pattern, then the LEX strategy is used to determine placement of the activation. Again, as with the CLIPS LEX strategy, negated patterns have pseudo time tags.

As an example, the following six activations have been listed in their MEA ordering (where the comma at the end of the activation indicates the presence of a negated pattern).

```
rule-2: f-3,f-1
rule-3: f-2,f-1
rule-6: f-1,f-4
rule-5: f-1,f-2,f-3,
rule-1: f-1,f-2,f-3
rule-4: f-1,f-2,
```

5.3.7 Random Strategy

Each activation is assigned a random number that is used to determine its placement among activations of equal salience. This random number is preserved when the strategy is changed so that the same ordering is reproduced when the random strategy is selected again (among activations that were on the agenda when the strategy was originally changed).

❖□□□□□□□□□□□□□

A conflict resolution strategy is an implicit mechanism for specifying the order in which rules of equal salience should be executed. In early expert system tools, this was often the only mechanism provided to specify the order. Because the mechanism is implicit, it's not possible to determine the programmer's original intent simply by looking at the code. [Of course in the real world there isn't a need to guess the original intent because the code is riddled with helpful comments.] Rather than explicitly indicating that rule A should be executed before rule B, the order of execution is implicitly determined by the order in which facts are asserted and the complexity of the rules. The assumption one must make when examining the code is that the original programmer carefully analyzed the rules and followed the necessary conventions so that the rules execute in the appropriate sequence.

Because they require explicit declarations, the preferred mechanisms in CLIPS for ordering the execution of rules are salience and modules. Salience allows one to explicitly specify that one rule should be executed before another rule. Modules allow one to explicitly specify that all of the rules in a particular group (module) should be executed before all of the rules in a different group. Thus, when designing a program the following convention should be followed: if two rules have the same salience, are in the same module, and are activated concurrently, then the order in which they are executed should not matter. For example, the following two rules need correction because they can be activated at the same time, but the order in which they execute matters:

```
(defrule rule-1
   (factoid a)
   =>
   (assert (factoid b)))

(defrule rule-2
   ?f <- (factoid a)
   (factoid d)
   =>
   (retract ?f)
   (assert (factoid c)))
```

Programmers should also be careful to avoid overusing salience. Trying to unravel the relationships between dozens of salience values can be just as confusing as the implicit use of a conflict resolution strategy in determining rule execution order. It's rarely necessary to use more than five to ten salience values in a well-designed program.

Most programs should use the default conflict resolution strategy of depth. The breadth, simplicity, and complexity strategies are provided largely for academic reasons (i.e. the study of 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 that 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 occurence 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).

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 several constraints connected together. In addition to literal constraints, CLIPS provides three other types of constraints: connective constraints, predicate constraints, and return value constraints. Wildcards are used within pattern CEs to indicate that a single field or group of fields can be matched by anything. Variables are used to store the value of a field so that it can be used later on the LHS of a rule in other conditional elements or on the RHS of a rule as an argument to an action.

The first field of any pattern *must* be a symbol and can not use any other constraints. This first field is used by CLIPS to determine if the pattern applies to an ordered fact, a template fact, or an instance. The symbol *object* is reserved to indicate an object pattern. Any other symbol used must correspond to a deftemplate name (or an implied deftemplate will be created). Slot names must also be symbols and cannot contain any other constraints.

For object and deftemplate patterns, a single field slot can only contain one field constraint and that field constraint must only be able to match a single field (no multifield wildcards or variables). A multifield slot can contain any number of field constraints.

The examples and syntax shown in the following sections will be for ordered and deftemplate fact patterns. Section 5.4.1.7 will discuss differences between deftemplate patterns and object patterns. The following constructs are used by the examples.

```
(deffacts color-facts
  (colors rgb primary red green blue)
  (colors rgb secondary cyan yellow magenta)
  (colors ryb primary red yellow blue)
  (colors ryb secondary purple orange green))

(deftemplate person
  (multislot name)
  (slot age))

(deffacts people
  (person (name Joe Bob Green) (age 20))
  (person (name Martin Brown) (age 20))
  (person (name Frank Martin) (age 34))
  (person (name Ann Green) (age 34))
  (person (name Sue Ann Brown) (age 20)))
```

5.4.1.1 Literal Constraints

The most basic constraint that can be used in a pattern CE is one which precisely defines the exact value that will match a field. This is called a **literal constraint**. A **literal pattern CE** consists entirely of constants such as floats, integers, symbols, strings, and instance names. It does not contain any variables or wildcards. All constraints in a literal pattern must be matched exactly by all fields of a pattern entity.

Syntax

An ordered pattern conditional element containing only literals has the following basic syntax:

```
(<constant-1> ... <constant-n>)
```

A deftemplate pattern conditional element containing only literals has the following basic syntax:

```
(<deftemplate-name> (<slot-name-1> <constant-1>)
                                   •
                                   •
                                   •
                    (<slot-name-n> <constant-n>))
```

Example 1

This example utilizes the *color-facts* deffacts shown in section 5.4.1.

```
CLIPS>
(defrule rgb-primary
   (colors rgb primary red green blue)
   =>)
CLIPS> (reset)
```

```
CLIPS> (agenda)
0      rgb-primary: f-1
For a total of 1 activation.
CLIPS> (facts)
f-1     (colors rgb primary red green blue)
f-2     (colors rgb secondary cyan yellow magenta)
f-3     (colors ryb primary red yellow blue)
f-4     (colors ryb secondary purple orange green)
For a total of 4 facts.
CLIPS>
```

Example 2

This example utilizes the *person* deftemplate and *people* deffacts shown in section 5.4.1.

```
CLIPS>
(defrule Find-Joe-Bob
   (person (name Joe Bob Green) (age 20))
   =>)
CLIPS>
(defrule Find-Ann
   (person (age 34) (name Ann Green))
   =>)
CLIPS> (reset)
CLIPS> (agenda)
0      Find-Ann: f-4
0      Find-Joe-Bob: f-1
For a total of 2 activations.
CLIPS> (facts)
f-1     (person (name Joe Bob Green) (age 20))
f-2     (person (name Martin Brown) (age 20))
f-3     (person (name Frank Martin) (age 34))
f-4     (person (name Ann Green) (age 34))
f-5     (person (name Sue Ann Brown) (age 20))
For a total of 5 facts.
CLIPS>
```

5.4.1.2 Wildcards Single- and Multifield

CLIPS has two **wildcard** symbols that may be used to match fields in a pattern. CLIPS interprets these wildcard symbols as standing in place of some part of a pattern entity. The **single-field wildcard**, denoted by a question mark character (?), matches any value stored in exactly one field in the pattern entity. The **multifield wildcard**, denoted by a dollar sign followed by a question mark ($?), matches any value in *zero* or more fields in a pattern entity. Single-field and multifield wildcards may be combined in a single pattern in any combination. It is illegal to use a multifield wildcard in a single field slot of a deftemplate or object pattern. By default, an unspecified single-field slot in a deftemplate/object pattern is matched against an implied single-field wildcard. Similarly, an unspecified multifield slot in a deftemplate/object pattern is matched against an implied multifield-wildcard.

Syntax

An ordered pattern conditional element containing only literals and wildcards has the following basic syntax:

```
(<constraint-1> ... <constraint-n>)
```

where

```
<constraint> ::= <constant> | ? | $?
```

A deftemplate pattern conditional element containing only literals and wildcards has the following basic syntax:

```
(<deftemplate-name> (<slot-name-1> <constraint-1>)
                                   •
                                   •
                                   •
                    (<slot-name-n> <constraint-n>))
```

Example 1

This example utilizes the *color-facts* deffacts shown in section 5.4.1.

```
CLIPS>
(defrule find-green
   (colors ? ? $? green $?)
   =>)
CLIPS> (reset)
CLIPS> (agenda)
0      find-green: f-4
0      find-green: f-1
For a total of 2 activations.
CLIPS> (facts)
f-1     (colors rgb primary red green blue)
f-2     (colors rgb secondary cyan yellow magenta)
f-3     (colors ryb primary red yellow blue)
f-4     (colors ryb secondary purple orange green)
For a total of 4 facts.
CLIPS>
```

Example 2

This example utilizes the *person* deftemplate and *people* deffacts shown in section 5.4.1.

```
CLIPS>
(defrule match-all-persons
   (person)
   =>)
CLIPS> (reset)
CLIPS> (agenda)
0      match-all-persons: f-5
0      match-all-persons: f-4
0      match-all-persons: f-3
```

```
0      match-all-persons: f-2
0      match-all-persons: f-1
For a total of 5 activations.
CLIPS>
```

Example 3

This example utilizes the *person* deftemplate and *people* deffacts shown in section 5.4.1.

```
CLIPS>
(defrule match-two-names
   (person (name ? ?))
   =>)
CLIPS>
(defrule match-three-names
   (person (name ? ? ?))
   =>)
CLIPS> (reset)
CLIPS> (agenda)
0      match-three-names: f-5
0      match-two-names: f-4
0      match-two-names: f-3
0      match-two-names: f-2
0      match-three-names: f-1
For a total of 5 activations.
CLIPS> (facts)
f-1     (person (name Joe Bob Green) (age 20))
f-2     (person (name Martin Brown) (age 20))
f-3     (person (name Frank Martin) (age 34))
f-4     (person (name Ann Green) (age 34))
f-5     (person (name Sue Ann Brown) (age 20))
For a total of 5 facts.
CLIPS>
```

Example 4

This example utilizes the *person* deftemplate and *people* deffacts shown in section 5.4.1.

```
CLIPS>
(defrule last-name-brown
   (person (name $? Brown))
   =>)
CLIPS>
(defrule name-contains-ann
   (person (name $? Ann $?))
   =>)
CLIPS> (reset)
CLIPS> (agenda)
0      last-name-brown: f-5
0      name-contains-ann: f-5
0      name-contains-ann: f-4
0      last-name-brown: f-2
For a total of 4 activations.
CLIPS> (facts)
f-1     (person (name Joe Bob Green) (age 20))
f-2     (person (name Martin Brown) (age 20))
f-3     (person (name Frank Martin) (age 34))
f-4     (person (name Ann Green) (age 34))
f-5     (person (name Sue Ann Brown) (age 20))
For a total of 5 facts.
```

```
CLIPS>
```

Multifield wildcard and literal constraints can be combined to yield some powerful pattern-matching capabilities. A pattern to match all of the facts that have the symbol YELLOW in any field (other than the first) could be written as

```
(data $? YELLOW $?)
```

Some examples of what this pattern would match are

```
(data YELLOW blue red green)
(data YELLOW red)
(data red YELLOW)
(data YELLOW)
(data YELLOW data YELLOW)
```

The last fact will match twice since YELLOW appears twice in the fact. The use of multifield wildcards should be confined to cases of patterns in which the single-field wildcard cannot create a pattern that satisfies the match required, since the multifield wildcard produces every possible match combination that can be derived from a pattern entity. This derivation of matches requires a significant amount of time to perform when compared to the time needed to perform a single-field match.

5.4.1.3 Variables Single- and Multifield

Wildcard symbols replace portions of a pattern and accept any value. The value of the field being replaced may be captured in a **variable** for comparison, display, or other manipulations. This is done by directly following the wildcard symbol with a variable name.

Syntax

Expanding on the syntax definition given in section 5.4.1.2 now gives:

```
<constraint> ::= <constant> | ? | $? |
                 <single-field-variable> |
                 <multifield-variable>

<single-field-variable> ::= ?<variable-symbol>

<multifield-variable>   ::= $?<variable-symbol>
```

where <variable-symbol> is similar to a symbol, except that it must start with an alphabetic character. Double quotes are not allowed as part of a variable name; i.e. a string cannot be used for a variable name. The rules for pattern-matching are similar to those for wildcard symbols. On its first appearance, a variable acts just like a wildcard in that it will bind to any value in the field(s). However, later appearances of the variable require the field(s) to match the binding of the variable. The binding will only be true within the scope of the rule in which it occurs. Each rule has a private list of variable names with their associated values; thus, variables are local to a rule.

Bound variables can be passed to external functions. The $ operator has special significance on the LHS as a pattern-matching operator to indicate that zero or more fields need to be matched. In other places (such as the RHS of a rule), the $ in front of a variable indicates that sequence expansion should take place before calling the function. Thus, when passed as parameters in function calls (either on the LHS or RHS of a rule), multifield variables should not be preceded by the $ (unless sequence expansion is desired). All other uses of a multifield variable on the LHS of a rule, however, should use the $. It is illegal to use a multifield variable in a single field slot of a deftemplate/object pattern.

Example 1

```
CLIPS> (clear)
CLIPS> (reset)
CLIPS> (assert (data 2 blue green)
               (data 1 blue)
               (data 1 blue red))
<Fact-3>
CLIPS> (facts)
f-1     (data 2 blue green)
f-2     (data 1 blue)
f-3     (data 1 blue red)
For a total of 3 facts.
CLIPS>
(defrule find-data-1
  (data ?x ?y ?z)
  =>
  (println ?x " : " ?y " : " ?z))
CLIPS> (run)
1 : blue : red
2 : blue : green
CLIPS>
```

Example 2

```
CLIPS> (reset)
CLIPS> (assert (data 1 blue)
               (data 1 blue red)
               (data 1 blue red 6.9))
<Fact-3>
CLIPS> (facts)
f-1     (data 1 blue)
f-2     (data 1 blue red)
f-3     (data 1 blue red 6.9)
For a total of 3 facts.
CLIPS>
(defrule find-data-1
  (data ?x $?y ?z)
  =>
  (print "?x = " ?x crlf
         "?y = " ?y crlf
         "?z = " ?z crlf
         "------" crlf))
CLIPS> (run)
?x = 1
?y = (blue red)
?z = 6.9
------
?x = 1
```

```
?y = (blue)
?z = red
------
?x = 1
?y = ()
?z = blue
------
CLIPS>
```

Once the initial binding of a variable occurs, all references to that variable have to match the value that the first binding matched. This applies to both single- and multifield variables. It also applies across patterns.

Example 3

```
CLIPS> (clear)
CLIPS>
(deffacts data
  (data red green)
  (data purple blue)
  (data purple green)
  (data red blue green)
  (data purple blue green)
  (data purple blue brown))
CLIPS>
(defrule find-data-1
  (data red ?x)
  (data purple ?x)
  =>)
CLIPS>
(defrule find-data-2
  (data red $?x)
  (data purple $?x)
  =>)
CLIPS> (reset)
CLIPS> (facts)
f-1     (data red green)
f-2     (data purple blue)
f-3     (data purple green)
f-4     (data red blue green)
f-5     (data purple blue green)
f-6     (data purple blue brown)
For a total of 6 facts.
CLIPS> (agenda)
0      find-data-2: f-4,f-5
0      find-data-1: f-1,f-3
0      find-data-2: f-1,f-3
For a total of 3 activations.
CLIPS>
```

5.4.1.4 Connective Constraints

Three **connective constraints** are available for connecting individual constraints and variables to each other. These are the & (and), | (or), and ~ (not) connective constraints. The & constraint is satisfied if the two adjoining constraints are satisfied. The | constraint is satisfied if either of the two adjoining constraints is satisfied. The ~ constraint is satisfied if the following constraint is

not satisfied. The connective constraints can be combined in almost any manner or number to constrain the value of specific fields while pattern-matching. The ~ constraint has highest precedence, followed by the & constraint, followed by the | constraint. Otherwise, evaluation of multiple constraints can be considered to occur from left to right. There is one exception to the precedence rules that applies to the binding occurrence of a variable. If the first constraint is a variable followed by an & connective constraint, then the first constraint is treated as a separate constraint which also must be satisified. Thus the constraint ?x&red|blue is treated like ?x&(red|blue) rather than (?x&red)|blue as the normal precedence rules would indicate.

Basic Syntax

Connective constraints have the following basic syntax:

```
<term-1>&<term-2> ... &<term-3>

<term-1>|<term-2> ... |<term-3>

~<term>
```

where <term> could be a single-field variable, multifield variable, constant, or connected constraint.

Syntax

Expanding on the syntax definition given in section 5.4.1.3 now gives:

```
<constraint> ::= ? | $? | <connected-constraint>

<connected-constraint>
             ::= <single-constraint> |
                 <single-constraint> & <connected-constraint> |
                 <single-constraint> | <connected-constraint>

<single-constraint> ::= <term> | ~<term>

<term> ::= <constant> |
           <single-field-variable> |
           <multifield-variable>
```

The & constraint typically is used only in conjunction with other constraints or variable bindings. Notice that connective constraints may be used together and/or with variable bindings. If the first term of a connective constraint is the first occurrence of a variable name, then the field will be constrained only by the remaining field constraints. The variable will be bound to the value of the field. If the variable has been bound previously, it is considered an additional constraint along with the remaining field constraints; i.e., the field must have the same value already bound to the variable and must satisfy the field constraints.

Example 1

```
CLIPS> (clear)
CLIPS> (deftemplate data-B (slot value))
CLIPS>
(deffacts AB
  (data-A green)
  (data-A blue)
  (data-B (value red))
  (data-B (value blue)))
CLIPS>
(defrule example1-1
  (data-A ~blue)
  =>)
CLIPS>
(defrule example1-2
  (data-B (value ~red&~green))
  =>)
CLIPS>
(defrule example1-3
  (data-B (value green|red))
  =>)
CLIPS> (reset)
CLIPS> (facts)
f-1     (data-A green)
f-2     (data-A blue)
f-3     (data-B (value red))
f-4     (data-B (value blue))
For a total of 4 facts.
CLIPS> (agenda)
0      example1-2: f-4
0      example1-3: f-3
0      example1-1: f-1
For a total of 3 activations.
CLIPS>
```

Example 2

```
CLIPS> (clear)
CLIPS> (deftemplate data-B (slot value))
CLIPS>
(deffacts B
  (data-B (value red))
  (data-B (value blue)))
CLIPS>
(defrule example2-1
  (data-B (value ?x&~red&~green))
  =>
  (println "?x in example2-1 = " ?x))
CLIPS>
(defrule example2-2
  (data-B (value ?x&green|red))
  =>
  (println "?x in example2-2 = " ?x))
CLIPS> (reset)
CLIPS> (run)
?x in example2-1 = blue
?x in example2-2 = red
CLIPS>
```

Example 3

```
CLIPS> (clear)
CLIPS> (deftemplate data-B (slot value))
CLIPS>
(deffacts AB
  (data-A green)
  (data-A blue)
  (data-B (value red))
  (data-B (value blue)))
CLIPS>
(defrule example3-1
  (data-A ?x&~green)
  (data-B (value ?y&~?x))
  =>)
CLIPS>
(defrule example3-2
  (data-A ?x)
  (data-B (value ?x&green|blue))
  =>)
CLIPS>
(defrule example3-3
  (data-A ?x)
  (data-B (value ?y&blue|?x))
  =>)
CLIPS> (reset)
CLIPS> (facts)
f-1     (data-A green)
f-2     (data-A blue)
f-3     (data-B (value red))
f-4     (data-B (value blue))
For a total of 4 facts.
CLIPS> (agenda)
0      example3-3: f-1,f-4
0      example3-3: f-2,f-4
0      example3-2: f-2,f-4
0      example3-1: f-2,f-3
For a total of 4 activations.
CLIPS>
```

5.4.1.5 Predicate Constraints

Sometimes it becomes necessary to constrain a field based upon the truth of a given boolean expression. CLIPS allows the use of a **predicate constraint** to restrict a field in this manner. The predicate constraint allows a **predicate function** (one returning the symbol FALSE for unsatisfied and a non-FALSE value for satisfied) to be called during the pattern-matching process. If the predicate function returns a non-FALSE value, the constraint is satisfied. If the predicate function returns the symbol FALSE, the constraint is not satisfied. A predicate constraint is invoked by following a colon with an appropriate function call to a predicate function. Typically, predicate constraints are used in conjunction with a connective constraint and a variable binding (i.e. you have to bind the variable to be tested and then connect it to the predicate constraint).

Basic Syntax

```
:<function-call>
```

Syntax

Expanding on the syntax definition given in section 5.4.1.4 now gives:

```
<term> ::= <constant> |
           <single-field-variable> |
           <multifield-variable> |
           :<function-call>
```

Multiple predicate constraints may be used to constrain a single field. CLIPS provides several predicate functions (see section 12.1). Users also may develop their own predicate functions.

Example 1

```
CLIPS> (clear)
CLIPS>
(deftemplate person
   (slot name)
   (slot age))
CLIPS>
(defrule adult
   (person (age ?age&:(>= ?age 18)))
   =>)
CLIPS>
(assert (person (name John) (age 20))  ; f-1
        (person (name Sally) (age 18)) ; f-2
        (person (name Bill) (age 14))) ; f-3
<Fact-3>
CLIPS> (agenda)
0      adult: f-2
0      adult: f-1
For a total of 2 activations.
CLIPS>
```

Example 2

```
CLIPS> (clear)
CLIPS>
(deftemplate person
   (slot name)
   (multislot attributes))
CLIPS>
(defrule not-tall
   (person (attributes $?a&~:(member$ tall ?a)))
   =>)
CLIPS>
(assert (person (name John) (attributes tall thin))   ; f-1
        (person (name Greg) (attributes short stout)) ; f-2
        (person (name Jill) (attributes young tall))) ; f-3
<Fact-3>
CLIPS> (agenda)
0      not-tall: f-2
For a total of 1 activation.
CLIPS>
```

Example 3

```
CLIPS> (clear)
CLIPS>
(deftemplate person
   (slot name)
   (slot age))
CLIPS>
(defrule teenager
   (person (age ?age&:(>= ?age 13)&:(<= ?age 19)))
   =>)
CLIPS>
(assert (person (name John) (age 20))  ; f-1
        (person (name Sally) (age 18)) ; f-2
        (person (name Bill) (age 14))) ; f-3
<Fact-3>
CLIPS> (agenda)
0      teenager: f-3
0      teenager: f-2
For a total of 2 activations.
CLIPS>
```

Example 4

```
CLIPS> (clear)
CLIPS>
(deftemplate person
   (slot name)
   (slot age))
CLIPS>
(defrule older
   (person (age ?age1))
   (person (age ?age2&:(> ?age1 ?age2)))
   =>)
CLIPS>
(assert (person (name John) (age 20))  ; f-1
        (person (name Sally) (age 18)) ; f-2
        (person (name Bill) (age 14))) ; f-3
<Fact-3>
CLIPS> (agenda)
0      older: f-1,f-3
0      older: f-2,f-3
0      older: f-1,f-2
For a total of 3 activations.
CLIPS>
```

Example 5

```
CLIPS> (clear)
CLIPS>
(deftemplate person
   (slot name)
   (multislot siblings))
CLIPS>
(defrule large-family
   (person (siblings $?s&:(> (length$ ?s) 4)))
   =>)
CLIPS>
(assert (person (name John) (siblings Fred Gwen))             ; f-1
        (person (name Greg))                                  ; f-2
        (person (name Jill) (siblings Joe Sue Lou Mark Dot))) ; f-3
```

```
<Fact-3>
CLIPS> (agenda)
0      large-family: f-3
For a total of 1 activation.
CLIPS>
```

5.4.1.6 Return Value Constraints

It is possible to use the return value of an external function to constrain the value of a field. The **return value constraint** (=) allows the user to call external functions from inside a pattern. (This constraint is different from the comparison function that uses the same symbol. The difference can be determined from context.) The return value must be one of the primitive data types. This value is incorporated directly into the pattern at the position at which the function was called as if it were a literal constraint, and any matching patterns must match this value as though the rule were typed with that value. Note that the function is evaluated each time the constraint is checked (not just once).

Basic Syntax

```
=<function-call>
```

Syntax

Expanding on the syntax definition given in section 5.4.1.5 now gives:

```
<term> ::= <constant> |
           <single-field-variable> |
           <multifield-variable> |
           :<function-call> |
           =<function-call>
```

Example 1

```
CLIPS> (clear)
CLIPS> (deftemplate data (slot x) (slot y))
CLIPS>
(defrule twice
  (data (x ?x) (y =(* 2 ?x)))
  =>)
CLIPS> (assert (data (x 2) (y 4))  ; f-1
               (data (x 3) (y 9))) ; f-2
<Fact-1>
CLIPS> (agenda)
0      twice: f-1
For a total of 1 activation.
CLIPS>
```

Example 2

```
CLIPS> (clear)
CLIPS>
(defclass DATA (is-a USER)
   (slot x))
CLIPS>
```

```
(defrule return-value-example-2
   (object (is-a DATA)
           (x ?x1))
   (object (is-a DATA)
           (x ?x2&=(+ 5 ?x1)|=(- 12 ?x1)))
   =>)
CLIPS> (make-instance of DATA (x 4))
[gen1]
CLIPS> (make-instance of DATA (x 9))
[gen2]
CLIPS> (make-instance of DATA (x 3))
[gen3]
CLIPS> (agenda)
0      return-value-example-2: [gen3],[gen2]
0      return-value-example-2: [gen2],[gen3]
0      return-value-example-2: [gen1],[gen2]
For a total of 3 activations.
CLIPS>
```

5.4.1.7 Pattern-Matching with Object Patterns

Instances of user-defined classes in COOL can be pattern-matched on the left-hand side of rules. Patterns can only match objects for which the object's most specific class is defined before the pattern and which are in scope for the current module. Any classes that could have objects that match the pattern cannot be deleted or changed until the pattern is deleted. Even if a rule is deleted by its RHS, the classes bound to its patterns cannot be changed until after the RHS finishes executing.

When an instance is created or deleted, all patterns applicable to that object are updated. However, when a slot is changed, only those patterns that explicitly match on that slot are affected. Thus, one could use logical dependencies to hook to a change to a particular slot (rather than a change to any slot, which is all that is possible with deftemplates).

Changes to non-reactive slots or instances of non-reactive classes (see sections 9.3.2.2 and 9.3.3.7) will have no effect on rules. Also Rete network activity will not be immediately apparent after changes to slots are made if pattern-matching is being delayed through the use of the **make-instance**, **initialize-instance**, **modify-instance**, **message-modify-instance**, **duplicate-instance**, **message-duplicate-instance** or **object-pattern-match-delay** functions.

Syntax

```
<object-pattern>       ::= (object <attribute-constraint>*)

<attribute-constraint> ::= (is-a <constraint>) |
                           (name <constraint>) |
                           (<slot-name> <constraint>*)
```

The **is-a** constraint is used for specifying class constraints such as "Is this object a member of class FOO?". The is-a constraint also encompasses subclasses of the matching classes unless specifically excluded by the pattern. The **name** constraint is used for specifying a specific

instance on which to pattern-match. The evaluation of the name constraint must be of primitive type **instance-name**, not **symbol**. Multifield constraints (such as $?) cannot be used with the is-a or name constraints. Other than these special cases, constraints used in object slots work similarly to constraints used in deftemplate slots. As with deftemplate patterns, slot names for object patterns must be symbols and can not contain any other constraints.

Example 1

The following rules illustrate pattern-matching on an object's class.

```
(defrule class-match-1
  (object)
  =>)

(defrule class-match-2
  (object (is-a FOO))
  =>)

(defrule class-match-3
  (object (is-a FOO | BAR))
  =>)

(defrule class-match-4
  (object (is-a ?x))
  (object (is-a ~?x))
  =>)
```

Rule *class-match-1* is satisified by all instances of any reactive class. Rule *class-match-2* is satisfied by all instances of class FOO. Rule *class-match-3* is satisfied by all instances of class FOO or BAR. Rule *class-match-4* will be satisfied by any two instances of mutually exclusive classes.

Example 2

The following rules illustrate pattern-matching on various attributes of an object's slots.

```
(defrule slot-match-1
  (object (width))
  =>)

(defrule slot-match-2
  (object (width ?))
  =>)

(defrule slot-match-3
  (object (width $?))
  =>)
```

Rule *slot-match-1* is satisfied by all instances of reactive classes that contain a reactive *width* slot with a zero length multifield value. Rule *slot-match-2* is satisfied by all instances of reactive classes that contain a reactive single or multifield *width* slot that is bound to a single value. Rule *slot-match-3* is satisfied by all instances of reactive classes that contain a reactive single or multifield *width* slot that is bound to any number of values. Note that a slot containing a zero

length multifield value would satisfy rules *slot-match-1* and *slot-match-3*, but not rule *slot-match-2* (because the value's cardinality is zero).

Example 3

The following rules illustrate pattern-matching on the slot values of an object.

```
(defrule value-match-1
  (object (width 10)
  =>)

(defrule value-match-2
  (object (width ?x&:(> ?x 20)))
  =>)

(defrule value-match-3
  (object (width ?x) (height ?x))
  =>)
```

Rule *value-match-1* is satisified by all instances of reactive classes that contain a reactive *width* slot with value 10. Rule *value-match-2* is satisfied by all instances of reactive classes that contain a reactive *width* slot that has a value greater than 20. Rule *value-match-3* is satisfied by all instances of reactive classes that contain a reactive *width* and *height* slots with the same value.

5.4.1.8 Pattern-Addresses

Certain RHS actions, such as **retract** and **unmake-instance**, operate on an entire pattern CE. To signify which fact or instance they are to act upon, a variable can be bound to the **fact-address** or **instance-address** of a pattern CE. Collectively, fact-addresses and instance-addresses bound on the LHS of a rule are referred to as **pattern-addresses**.

Syntax

```
<assigned-pattern-CE> ::= ?<variable-symbol> <- <pattern-CE>
```

The left arrow, **<-**, is a required part of the syntax. A variable bound to a fact-address or instance-address can be compared to other variables or passed to external functions. Variables bound to a fact or instance-address may later be used to constrain fields within a pattern CE, however, the reverse is not allowed. It is an error to bind a varible to a **not** CE.

Examples

```
(defrule dummy
   (data 1)
   ?fact <- (dummy pattern)
   =>
   (retract ?fact))

(defrule compare-facts-1
   ?f1 <- (color ~red)
   ?f2 <- (color ~green)
```

```
   (test (neq ?f1 ?f2))
   =>
   (println "Rule fires from different facts"))

(defrule compare-facts-2
   ?f1 <- (color ~red)
   ?f2 <- (color ~green&:(neq ?f1 ?f2))
   =>
   (println "Rule fires from different facts"))

(defrule print-and-delete-all-objects
   ?ins <- (object)
   =>
   (send ?ins print)
   (unmake-instance ?ins))
```

5.4.2 Test Conditional Element

Field constraints used within pattern CEs allow very descriptive constraints to be applied to pattern-matching. Additional capability is provided with the **test conditional element**. The test CE is satisfied if the function call within the test CE evaluates to a non-FALSE value and unsatisfied if the function call evaluates to FALSE. As with predicate constraints, the user can compare the variable bindings that already have occurred in any manner. Mathematical comparisons on variables (e.g., is the difference between ?x and ?y greater than some value?) and complex logical or equality comparisons can be done. External functions also can be called which compare variables in any way that the user desires.

Any kind of external function may be embedded within a **test** conditional element (or within field constraints). User-defined predicate functions must take arguments as defined in the *Advanced Programming Guide*. CLIPS provides several predicate functions (see section 12.1).

Syntax

```
<test-CE> ::= (test <function-call>)
```

Since the symbol **test** is used to indicate this type of conditional element, rules may not use the symbol test as the first field in a pattern CE. A **test** CE is evaluated when all proceeding CEs are satisfied. This means that a **test** CE will be evaluated more than once if the proceeding CEs can be satisfied by more than one group of pattern entities. In order to cause the reevaluation of a **test** CE, a pattern entity matching a CE prior to the **test** CE must be changed.

Example 1

This example checks to see if the difference between two numbers is greater than or equal to three:

```
CLIPS> (clear)
CLIPS>
(defrule example-1
  (data ?x)
```

```
   (value ?y)
   (test (>= (abs (- ?y ?x)) 3))
   =>)
CLIPS> (assert (data 6) (value 9))
<Fact-2>
CLIPS> (agenda)
0      example-1: f-1,f-2
For a total of 1 activation.
CLIPS>
```

Example 2

This example checks to see if there is a positive slope between two points on a line.

```
CLIPS> (clear)
CLIPS>
(deffunction positive-slope
   (?x1 ?y1 ?x2 ?y2)
   (< 0 (/ (- ?y2 ?y1) (- ?x2 ?x1))))
CLIPS>
(defrule example-2
   (point ?a ?x1 ?y1)
   (point ?b ?x2 ?y2)
   (test (> ?b ?a))
   (test (positive-slope ?x1 ?y1 ?x2 ?y2))
   =>)
CLIPS>
(assert (point 1 4.0 7.0) (point 2 5.0 9.0))
<Fact-2>
CLIPS> (agenda)
0      example-2: f-1,f-2
For a total of 1 activation.
CLIPS>
```

5.4.3 Or Conditional Element

The **or conditional element** allows any one of several conditional elements to activate a rule. If any of the conditional elements inside of the **or** CE is satisfied, then the **or** CE is satisfied. If all other LHS conditional elements are satisfied, the rule will be activated. Note that a rule will be activated for each conditional element with an **or** CE that is satisfied (assuming the other conditional elements of the rule are also satisfied). Any number of conditional elements may appear within an **or** CE. The **or** CE produces the identical effect of writing several rules with similar LHS's and RHS's.

Syntax

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

Again, if more than one of the conditional elements in the **or** CE can be met, the rule will fire *multiple times*, once for each satisfied combination of conditions.

Example

```
(defrule system-fault
  (error-status unknown)
  (or (temp high)
      (valve broken)
      (pump (status off)))
  =>
  (println "The system has a fault."))
```

Note that the above example is exactly equivalent to the following three (separate) rules:

```
(defrule system-fault
   (error-status unknown)
   (pump (status off))
   =>
   (println "The system has a fault."))

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

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

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. Note that the LHS of any rule is enclosed within an implied **and** CE.

Syntax

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

Example

```
(defrule system-flow
   (error-status confirmed)
   (or (and (temp high)
            (valve closed))
       (and (temp low)
            (valve open)))
```

```
   =>
   (println "The system is having a flow problem."))
```

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.

Examples

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

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

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

5.4.6 Exists Conditional Element

The **exists conditional element** provides a mechanism for determining if a group of specified CEs is satisfied by a least one set of pattern entities.

Syntax

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

The **exists** CE is implemented by replacing the **exists** keyword with two nested **not** CEs. For example, the following rule

```
(defrule example
   (exists (a ?x) (b ?x))
   =>)
```

is equivalent to the rule below

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

Because of the way the **exists** CE is implemented using **not** CEs, the restrictions which apply to CEs found within **not** CEs (such as binding a pattern CE to a fact-address) also apply to the CEs found within an **exists** CE.

Example

Given the following constructs,

```
CLIPS> (clear)
CLIPS>
(deftemplate hero
   (multislot name)
   (slot status (default unoccupied)))
CLIPS>
(deffacts goal-and-heroes
   (goal save-the-day)
   (hero (name Death Defying Man))
   (hero (name Stupendous Man))
   (hero (name Incredible Man)))
CLIPS>
(defrule save-the-day
   (goal save-the-day)
   (exists (hero (status unoccupied)))
   =>
   (println "The day is saved."))
CLIPS>
```

the following commands illustrate that even though there are three facts which can match the second CE in the *save-the-day* rule, there is only one partial match generated.

```
CLIPS> (reset)
CLIPS> (agenda)
0      save-the-day: f-1,*
For a total of 1 activation.
CLIPS> (facts)
f-1     (goal save-the-day)
f-2     (hero (name Death Defying Man) (status unoccupied))
f-3     (hero (name Stupendous Man) (status unoccupied))
f-4     (hero (name Incredible Man) (status unoccupied))
For a total of 4 facts.
CLIPS> (matches save-the-day)
```

```
Matches for Pattern 1
f-1
Matches for Pattern 2
f-2
f-3
f-4
Partial matches for CEs 1 - 2
f-1,*
Activations
f-1,*
(4 1 1)
CLIPS>
```

5.4.7 Forall Conditional Element

The **forall conditional element** provides a mechanism for determining if a group of specified CEs is satisfied for every occurence of another specified CE.

Syntax

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

The **forall** CE is implemented by replacing the **forall** keyword with combinations of **not** and **and** CEs. For example, the following rule

```
(defrule example
   (forall (a ?x) (b ?x) (c ?x))
   =>)
```

is equivalent to the rule below

```
(defrule example
   (not (and (a ?x)
             (not (and (b ?x) (c ?x)))))
   =>)
```

Because of the way the **forall** CE is implemented using **not** CEs, the restrictions which apply to CE found within **not** CEs (such as binding a pattern CE to a fact-address) also apply to the CEs found within an **forall** CE.

Example

The following rule determines if every student has passed in reading, writing, and arithmetic by using the **forall** CE.

```
CLIPS> (clear)
CLIPS>
(deftemplate student
   (slot name))
CLIPS>
(deftemplate passed
   (slot name)
```

```
   (slot subject))
CLIPS>
(defrule all-students-passed
   (forall
      (student (name ?name))
      (passed (name ?name) (subject reading))
      (passed (name ?name) (subject writing))
      (passed (name ?name) (subject arithmetic)))
   =>
   (println "All students passed."))
CLIPS>
```

The following commands illustrate how the **forall** CE works in the *all-students-passed* rule. Note that initially the *all-students-passed* rule is satisfied because there are no students.

```
CLIPS> (reset)
CLIPS> (agenda)
0      all-students-passed: *
For a total of 1 activation.
CLIPS>
```

After the (student Bob) fact is asserted, the rule is no longer satisfied since Bob has not passed reading, writing, and arithmetic.

```
CLIPS> (assert (student (name Bob))) ; f-1
<Fact-1>
CLIPS> (agenda)
CLIPS>
```

The rule is still not satisfied after Bob has passed reading and writing, since he still has not passed arithmetic.

```
CLIPS>
(assert (passed (name Bob) (subject reading))  ; f-2
        (passed (name Bob) (subject writing))) ; f-3
<Fact-3>
CLIPS> (agenda)
CLIPS>
```

Once Bob has passed arithmetic, the all-students-passed rule is reactivated.

```
CLIPS> (assert (passed (name Bob) (subject arithmetic))) ; f-4
<Fact-4>
CLIPS> (agenda)
0      all-students-passed: *
For a total of 1 activation.
CLIPS>
```

If a new student is asserted, then the rule is taken off the agenda, since John has not passed reading, writing, and arithmetic.

```
CLIPS> (assert (student (name John))) ; f-5
<Fact-5>
CLIPS> (agenda)
CLIPS>
```

Removing both *student* facts reactivates the rule again.

```
CLIPS> (retract 1 5)
CLIPS> (agenda)
0      all-students-passed: *
For a total of 1 activation.
CLIPS>
```

5.4.8 Logical Conditional Element

The **logical conditional element** provides a **truth maintenance** capability for pattern entities (facts or instances) created by rules that 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 that 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).

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** CEs. 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.

```
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS. (watch rules)
CLIPS> (assert (a) (b) (c) (d) (e) (f))
==> f-1     (a)
==> f-2     (b)
==> f-3     (c)
==> Activation 0      rule1: f-1,f-2,f-3
==> f-4     (d)
==> f-5     (e)
==> f-6     (f)
==> Activation 0      rule2: f-4,f-5,f-6
<Fact-6>
CLIPS> (run)
FIRE    1 rule2: f-4,f-5,f-6 ; 1st rule adds logical support
==> f-7     (g)
==> f-8     (h)
FIRE    2 rule1: f-1,f-2,f-3 ; 2nd rule adds further support
CLIPS> (retract 1)
```

```
<== f-1     (a)              ; Removes 1st support for (g) and (h)
CLIPS> (assert (h))          ; (h) is unconditionally supported
FALSE
CLIPS> (retract 4)
<== f-4     (d)              ; Removes 2nd support for (g)
<== f-7     (g)              ; (g) has no more support
CLIPS> (unwatch all)
CLIPS>
```

As mentioned in section 5.4.1.7, the logical CE can be used with an object pattern to create pattern entities that are logically dependent on changes to specific slots in the matching instance(s) rather than all slots. This cannot be accomplished with template facts because a change to a template fact slot actually involves the retraction of the old template fact and the assertion of a new one, whereas a change to an instance slot is done in place. The example below illustrates this behavior:

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot foo)
   (slot bar))
CLIPS>
(deftemplate A
  (slot foo)
  (slot bar))
CLIPS>
(defrule match-A-s
  (logical (object (is-a A) (foo ?))
           (A (foo ?)))
=>
  (assert (new-fact)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (assert (A))
<Fact-1>
CLIPS> (watch facts)
CLIPS> (run)
==> f-2     (new-fact)
CLIPS> (send [a] put-bar 100)
100
CLIPS> (agenda)
CLIPS> (modify 1 (bar 100))
<== f-1     (A (foo nil) (bar nil))
<== f-2     (new-fact)
==> f-1     (A (foo nil) (bar 100))
<Fact-3>
CLIPS> (agenda)
0      match-A-s: [a],f-1
For a total of 1 activation.
CLIPS> (run)
==> f-3     (new-fact)
CLIPS> (send [a] put-foo 100)
<== f-3     (new-fact)
100
CLIPS> (agenda)
0      match-A-s: [a],f-1
For a total of 1 activation.
CLIPS> (unwatch facts)
CLIPS>
```

5.4.9 Automatic Replacement of LHS CEs

Under certain circumstances, CLIPS will change the CEs specified in the rule LHS.

5.4.9.1 Or CEs Following Not CEs

If an *or* CE immediately follows a *not* CE, then the *not*/*or* CE combination is replaced with an *and*/*not* CE combination where each of the CEs contained in the original *or* CE is enclosed within a *not* CE and then all of the *not* CEs are enclosed within a single *and* CE. For example, the following rule

```
(defrule example
   (a ?x)
   (not (or (b ?x)
            (c ?x)))
   =>)
```

would be changed as follows.

```
(defrule example
   (a ?x)
   (and (not (b ?x))
        (not (c ?x)))
   =>)
```

5.4.10 Declaring Rule Properties

This feature allows the properties or characteristics of a rule to be defined. The characteristics are declared on the LHS of a rule using the **declare** keyword. A rule may only have one **declare** statement and it must appear before the first conditional element on the LHS (as shown in section 5.1).

Syntax

```
<declaration>    ::= (declare <rule-property>+)

<rule-property>  ::= (salience <integer-expression>) |
                     (auto-focus <boolean-symbol>)

<boolean-symbol> ::= TRUE | FALSE
```

5.4.10.1 The Salience Rule Property

The **salience** rule property allows the user to assign a priority to a rule. When multiple rules are in the agenda, the rule with the highest priority will fire first. The declared salience value should be an expression that evaluates to an an integer in the range -10000 to +10000. Salience expressions may freely reference global variables and other functions (however, you should

avoid using functions with side-effects). If unspecified, the salience value for a rule defaults to zero.

Example

```
(defrule test-1
   (declare (salience 99))
   (fire test-1)
   =>
   (println "Rule test-1 firing."))

(defrule test-2
   (declare (salience (+ ?*constraint-salience* 10)))
   (fire test-2)
   =>
   (println "Rule test-2 firing."))
```

Salience values can be evaluated at one of three times: when a rule is defined, when a rule is activated, and every cycle of execution (the latter two situations are referred to as **dynamic salience**). By default, salience values are only evaluated when a rule is defined. The **set-salience-evaluation** command can be used to change this behavior. Note that each salience evaluation method encompasses the previous method (i.e. if saliences are evaluated every cycle, then they are also evaluated when rules are activated or defined).

❖□□□□□□□□□□□□□□

Despite the large number of possible values, with good design there's rarely a need for more than five salience values in a simple program and ten salience values in a complex program. Defining the salience values as global variables allows you to specify and document the values used by your program in a centralized location and also makes it easier to change the salience of a group of rules sharing the same salience value:

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

(defglobal ?*low-priority* = -100)

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

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

5.4.10.2 The Auto-Focus Rule Property

The **auto-focus** rule property allows an automatic **focus** command to be executed whenever a rule becomes activated. If the auto-focus property for a rule is TRUE, then a focus command on the module in which the rule is defined is automatically executed whenever the rule is activated.

If the auto-focus property for a rule is FALSE, then no action is taken when the rule is activated. If unspecified, the auto-focus value for a rule defaults to FALSE.

Example

```
(defrule VIOLATIONS::bad-age
   (declare (auto-focus TRUE))
   (person (name ?name) (age ?x&:(< ?x 0)))
   =>
   (println ?name " has a bad age value."))
```

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 13.2.3), 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.

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*))
   =>)
```

Note that this rule will not necessarily be updated when the value of ?*x* is changed. For example, if ?*x* is 4 and the fact (fact 3) is added, then the rule is not satisfied. If the value of ?*x* is now changed to 2, the rule will not be activated.

Example

```
(defglobal
   ?*x* = 3
   ?*y* = ?*x*
   ?*z* = (+ ?*x* ?*y*)
   ?*q* = (create$ a b c))
```

❖□□□□□□□□□□□□□

The inappropriate use of globals within rules is quite often the first resort of beginning programmers who have reached an impasse in developing a program because they do not fully understand how rules and pattern-matching work. As it relates to this issue, the following sentence from the beginning of this section is important enough to repeat:

Global variables can be accessed as part of the pattern-matching process, but changing them does not invoke the pattern-matching process.

Facts and instances are the primary mechanism that should be used to pass information from one rule to another specifically because they *do* invoke pattern-matching. A change to a slot value of a fact or instance will trigger pattern-matching ensuring that a rule is aware of the current state of that fact or instance. Since a change to a global variable does not trigger pattern-matching, it is possible for a rule to remain activated based on a past value of a global variable that is undesirable in most situations.

It's worth pointing out that facts and instances are no less 'global' in nature than global variables. Just as a rule can access any global variable that's visible (i.e. it hasn't been hidden through the use of modules), so too can it access any fact or instance belonging to a deftemplate or defclass that's visible. In the case of a fact, one can either pattern-match for the fact on the LHS of a rule or use the fact-set query functions from the RHS of a rule. In the case of an instance, pattern-matching and instance-set query functions can also be used, and in addition an instance can be directly referenced by name just as a global variable can.

Common Problem

One of the more common situations in which it is tempting to use global variables is collecting a group of slot values from a fact. First attempts at rules to accomplish this task often loop

endlessly because of rules inadvertently retriggered by changes. For example, the following rule will loop endlessly because the new *collection* fact asserted will create an activation with the same *factoid* fact that was just added to the *collection* fact:

```
(defrule add-factoid
   (factoid ?data)
   ?c <- (collection $?collection)
   =>
   (retract ?c)
   (assert (collection ?collection ?data)))
```

This problem can be corrected by removing the *factoid* fact just added to the *collection* fact:

```
(defrule add-factoid
   ?f <- (factoid ?data)
   ?c <- (collection $?collection)
   =>
   (retract ?f ?c)
   (assert (collection ?collection ?data)))
```

Retracting the *factoid* facts, however, isn't a viable solution if these facts are needed by other rules. A non-destructive approach makes use of temporary facts created by a helper rule:

```
(defrule add-factoid-helper
   (factoid ?data)
   =>
   (assert (temp-factoid ?data)))

(defrule add-factoid
   ?f <- (temp-factoid ?data)
   ?c <- (collection $?collection)
   =>
   (retract ?f ?c)
   (assert (collection ?collection ?data)))
```

It certainly looks simpler, however, to use a global variable to collect the slot values:

```
(defglobal ?*collection* = (create$))

(defrule add-factoid
   (factoid ?data)
   =>
   (bind ?*collection* (create$ ?*collection* ?data)))
```

Again, the drawback to this approach is that changes to a global variable do not trigger pattern-matching, so in spite of its greater complexity the fact-based approach is still preferable.

Although it's important to understand how each of the previous approaches work, they're not practical solutions. If there are 1000 *factoid* facts, the add-factoid/add-factoid-helper rules will each fire 1000 times generating and retracting 2000 facts. The best solution is to use the fact-set query functions to iterate over all of the *factoid* facts and generate the *collection* fact as the result of a single rule firing:

```
(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)
   =>
   (println "The collected data is " ?data))
```

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

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*)))
   =>
   (println ?day " is invalid"))

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

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:

```
(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.

Section 7:
Deffunction Construct

With the **deffunction** construct, new functions may be defined directly in CLIPS. Deffunctions are equivalent in use to other functions; see section 2.3.2 for more detail on calling functions in CLIPS. The only differences between user-defined external functions and deffunctions are that deffunctions are written in CLIPS and executed by CLIPS interpretively and user-defined external functions are written in an external language, such as C, and executed by CLIPS directly. Also, deffunctions allow the addition of new functions without having to recompile and relink CLIPS.

A deffunction is comprised of five elements: 1) a name, 2) an optional comment, 3) a list of zero or more required parameters, 4) an optional wildcard parameter to handle a variable number of arguments and 5) a sequence of actions, or expressions, which will be executed in order when the deffunction is called.

Syntax

```
(deffunction <name> [<comment>]
   (<regular-parameter>* [<wildcard-parameter>])
   <action>*)

<regular-parameter>    ::= <single-field-variable>
<wildcard-parameter>   ::= <multifield-variable>
```

A deffunction must have a unique name different from all other system functions and generic functions. In particular, a deffunction cannot be overloaded like a system function (see section 8 for more detail). A deffunction must be declared prior to being called from another deffunction, generic function method, object message-handler, rule, or the top-level. The only exception is a self recursive deffunction.

A deffunction may accept *exactly* or *at least* a specified number of arguments, depending on whether a wildcard parameter is used or not. The regular parameters specify the minimum number of arguments that must be passed to the deffunction. Each of these parameters may be referenced like a normal single-field variable within the actions of the deffunction. If a wildcard parameter is present, the deffunction may be passed any number of arguments greater than or equal to the minimum. If no wildcard parameter is present, then the deffunction must be passed exactly the number of arguments specified by the regular parameters. All arguments to a deffunction that do not correspond to a regular parameter are grouped into a multifield value that can be referenced by the wildcard parameter. The standard CLIPS multifield functions, such as length and nth, can be applied to the wildcard parameter.

Example

```
CLIPS> (clear)
CLIPS>
(deffunction print-args (?a ?b $?c)
   (println ?a " " ?b " and " (length ?c) " extras: " ?c))
CLIPS> (print-args 1 2)
1 2 and 0 extras: ()
CLIPS> (print-args a b c d)
a b and 2 extras: (c d)
CLIPS>
```

When a deffunction is called, its actions are executed in order. The return value of a deffunction is the evaluation of the last action. If a deffunction has no actions, its return value is the symbol FALSE. If an error occurs while the deffunction is executing, any actions not yet executed will be aborted, and the deffunction will return the symbol FALSE.

Deffunctions may be self and mutually recursive. Self recursion is accomplished simply by calling the deffunction from within its own actions.

Example

```
(deffunction factorial (?a)
   (if (or (not (integerp ?a)) (< ?a 0))
      then
      (println "Factorial Error!")
      else
      (if (= ?a 0)
         then 1
         else (* ?a (factorial (- ?a 1))))))
```

Mutual recursion between two deffunctions requires a forward declaration of one of the deffunctions. A forward declaration is simply a declaration of the deffunction without any actions. In the following example, the deffunction foo is forward declared so that it may be called by the deffunction bar. Then the deffunction foo is redefined with actions that call the deffunction bar.

Example

```
(deffunction foo ())

(deffunction bar ()
   (foo))

(deffunction foo ()
   (bar))
```

Section 8:
Generic Functions

With the **defgeneric** and **defmethod** constructs, new generic functions may be written directly in CLIPS. Generic functions are similar to deffunctions because they can be used to define new procedural code directly in CLIPS, and they can be called like any other function (see sections 2.3.2 and 7). However, generic functions are much more powerful because they can do different things depending on the types (or classes) and number of their arguments. For example, a "+" operator could be defined which performs concatenation for strings but still performs arithmetic addition for numbers. Generic functions are comprised of multiple components called **methods**, where each method handles different cases of arguments for the generic function. A generic function which has more than one method is said to be **overloaded**.

Generic functions can have system functions and user-defined external functions as *implicit* methods. For example, an overloading of the "+" operator to handle strings consists of two methods: 1) an implicit one which is the system function handling numerical addition, and 2) an explicit (user-defined) one handling string concatenation. Deffunctions, however, may not be methods of generic functions because they are subsumed by generic functions anyway. Deffunctions are only provided so that basic new functions can be added directly in CLIPS without the concerns of overloading. For example, a generic function that has only one method that restricts only the number of arguments is equivalent to a deffunction.

In most cases, generic function methods are not called directly (the function **call-specific-method** described in section 12.15.8 can be used to do so, however). CLIPS recognizes that a function call is generic and uses the generic function's arguments to find and execute the appropriate method. This process is termed the **generic dispatch**.

8.1 Note on the Use of the Term *Method*

Most OOP systems support procedural behavior of objects either through message-passing (e.g. Smalltalk) or by generic functions (e.g. CLOS). CLIPS supports both of these mechanisms, although generic functions are not strictly part of COOL. A generic function may examine the classes of its arguments but must still use messages within the bodies of its methods to manipulate any arguments that are instances of user-defined classes. Section 9 gives more details on COOL. The fact that CLIPS supports both mechanisms leads to confusion in terminology. In OOP systems that support message-passing only, the term **method** is used to denote the different implementations of a **message** for different classes. In systems that support generic functions only, however, the term **method** is used to denote the different implementations of a generic function for different sets of argument restrictions. To avoid this confusion, the term **message-handler** is used to take the place of **method** in the context of messages. Thus in

CLIPS, **message-handlers** denote the different implementations of a **message** for different classes, and **methods** denote the different implementations of a **generic function** for different sets of argument restrictions.

8.2 Performance Penalty of Generic Functions

A call to a generic function is computationally more expensive than a call to a system function, user-defined external function or deffunction. This is because CLIPS must first examine the function arguments to determine which method is applicable. A performance penalty of 15%-20% is not unexpected. Thus, generic functions should not be used for routines for which time is critical, such as routines that are called within a while loop, if at all possible. Also, generic functions should always have at least two methods. Deffunctions or user-defined external functions should be used when overloading is not required. A system or user-defined external function that is not overloaded will, of course, execute as quickly as ever, since the generic dispatch is unnecessary.

8.3 Order Dependence of Generic Function Definitions

If a construct which uses a system or user-defined external function is loaded before a generic function that uses that function as an implicit method, all calls to that function from that construct will bypass the generic dispatch. For example, if a generic function which overloads the "+" operator is defined after a rule which uses the "+" operator, that rule will always call the "+" system function directly. However, similar rules defined after the generic function will use the generic dispatch.

8.4 Defining a New Generic Function

A generic function is comprised of a header (similar to a forward declaration) and zero or more methods. A generic function header can either be explicitly declared by the user or implicitly declared by the definition of at least one method. A method is comprised of six elements: 1) a name (which identifies to which generic function the method belongs), 2) an optional index, 3) an optional comment, 4) a set of parameter **restrictions**, 5) an optional wildcard parameter restriction to handle a variable number of arguments and 6) a sequence of actions, or expressions, which will be executed in order when the method is called. The parameter restrictions are used by the generic dispatch to determine a method's applicability to a set of arguments when the generic function is actually called. The **defgeneric** construct is used to specify the generic function header, and the **defmethod** construct is used for each of the generic function's methods.

Syntax

```
(defgeneric <name> [<comment>])

(defmethod <name> [<index>] [<comment>]
   (<parameter-restriction>* [<wildcard-parameter-restriction>])
   <action>*)

<parameter-restriction> ::=
                        <single-field-variable> |
                        (<single-field-variable> <type>* [<query>])

<wildcard-parameter-restriction> ::=
                        <multifield-variable> |
                        (<multifield-variable> <type>* [<query>])

<type>    ::= <class-name>
<query>   ::= <global-variable> |
                     <function-call>
```

A generic function must be declared, either by a header or a method, prior to being called from another generic function method, deffunction, object message-handler, rule, or the top-level. The only exception is a self recursive generic function.

8.4.1 Generic Function Headers

A generic function is uniquely identified by name. In order to reference a generic function in other constructs before any of its methods are declared, an explicit header is necessary. Otherwise, the declaration of the first method implicitly creates a header. For example, two generic functions whose methods mutually call the other generic function (mutually recursive generic functions) would require explicit headers.

8.4.2 Method Indices

A method is uniquely identified by name and index, or by name and parameter restrictions. Each method for a generic function is assigned a unique integer index within the group of all methods for that generic function. Thus, if a new method is defined which has exactly the same name and parameter restrictions as another method, CLIPS will automatically replace the older method. However, any difference in parameter restrictions will cause the new method to be defined *in addition to* the older method. To replace an old method with one that has different parameter restrictions, the index of the old method can be explicitly specified in the new method definition. However, the parameter restrictions of the new method must not match that of another method with a different index. If an index is not specified, CLIPS assigns an index that has never been used by any method (past or current) of this generic function. The index assigned by CLIPS can be determined with the **list-defmethods** command.

8.4.3 Method Parameter Restrictions

Each parameter for a method can be defined to have arbitrarily complex restrictions or none at all. A parameter restriction is applied to a generic function argument at run-time to determine if a particular method will accept the argument. A parameter can have two types of restrictions: type and query. A type restriction constrains the classes of arguments that will be accepted for a parameter. A query restriction is a user-defined boolean test which must be satisfied for an argument to be acceptable. The complexity of parameter restrictions directly affects the speed of the generic dispatch.

A parameter that has no restrictions means that the method will accept any argument in that position. However, each method of a generic function must have parameter restrictions that make it distinguishable from all of the other methods so that the generic dispatch can tell which one to call at run-time. If there are no applicable methods for a particular generic function call, CLIPS will generate an error.

A type restriction allows the user to specify a list of types (or classes), one of which must match (or be a superclass of) the class of the generic function argument. If COOL is not installed in the current CLIPS configuration, the only types (or classes) available are: OBJECT, PRIMITIVE, LEXEME, SYMBOL, STRING, NUMBER, INTEGER, FLOAT, MULTIFIELD, FACT-ADDRESS and EXTERNAL-ADDRESS. Section 9 describes each of these system classes. With COOL, INSTANCE, INSTANCE-ADDRESS, INSTANCE-NAME, USER, and any user-defined classes are also available. Generic functions that use only the first group of types in their methods will work the same whether COOL is installed or not. The classes in a type restriction must be defined already, since they are used to predetermine the precedence between a generic function's methods. Redundant classes are not allowed in restriction class lists. For example, the following method parameter's type restriction is redundant since INTEGER is a subclass of NUMBER.

Example

```
(defmethod foo ((?a INTEGER NUMBER)))
```

If the type restriction (if any) is satisfied for an argument, then a query restriction (if any) will be applied. The query restriction must either be a global variable or a function call. CLIPS evaluates this expression, and if it evaluates to anything but the symbol FALSE, the restriction is considered satisfied. Since a query restriction is not always satisfied, queries should *not* have any side-effects, for they will be evaluated for a method that may not end up being applicable to the generic function call. Since parameter restrictions are examined from left to right, queries that involve multiple parameters should be included with the rightmost parameter. This insures that all parameter type restrictions have already been satisfied. For example, the following method delays evaluation of the query restriction until the classes of both arguments have been verified.

Example

```
(defmethod foo ((?a INTEGER) (?b INTEGER (> ?a ?b))))
```

If the argument passes all these tests, it is deemed acceptable to a method. If *all* generic function arguments are accepted by a method's restrictions, the method itself is deemed **applicable** to the set of arguments. When more than one method is applicable to a set of arguments, the generic dispatch must determine an ordering among them and execute the first one in that ordering. Method precedence is used for this purpose and will be discussed in section 8.5.2.

Example

In the following example, the first call to the generic function "+" executes the system operator "+", an implicit method for numerical addition. The second call executes the explicit method for string concatenation, since there are two arguments and they are both strings. The third call generates an error because the explicit method for string concatenation only accepts two arguments and the implicit method for numerical addition does not accept strings at all.

```
CLIPS> (clear)
CLIPS>
(defmethod + ((?a STRING) (?b STRING))
   (str-cat ?a ?b))
CLIPS> (+ 1 2)
3
CLIPS> (+ "foo" "bar")
"foobar"
CLIPS> (+ "foo" "bar" "woz")
[GENRCEXE1] No applicable methods for +.
FALSE
CLIPS>
```

8.4.4 Method Wildcard Parameter

A method may accept *exactly* or *at least* a specified number of arguments, depending on whether a wildcard parameter is used or not. The regular parameters specify the minimum number of arguments that must be passed to the method. Each of these parameters may be referenced like a normal single-field variable within the actions of the method. If a wildcard parameter is present, the method may be passed any number of arguments greater than or equal to the minimum. If no wildcard parameter is present, then the method must be passed exactly the number of arguments specified by the regular parameters. Method arguments that do not correspond to a regular parameter can be grouped into a multifield value that can be referenced by the wildcard parameter within the body of the method. The standard CLIPS multifield functions, such as length$ and expand$, can be applied to the wildcard parameter.

If multifield values are passed as extra arguments, they will all be merged into one multifield value referenced by the wildcard parameter. This is because CLIPS does not support nested multifield values.

Type and query restrictions can be applied to arguments grouped in the wildcard parameter similarly to regular parameters. Such restrictions apply to each individual field of the resulting multifield value (not the entire multifield). However, expressions involving the wildcard parameter variable may be used in the query. In addition, a special variable may be used in query restrictions on the wildcard parameter to refer to the individual arguments grouped into the wildcard: **?current-argument**. This variable is only in scope within the query and has no meaning in the body of the method. For example, to create a version of the '+' operator which acts differently for sums of all even integers:

Example

```
CLIPS>
(defmethod +
  (($?any INTEGER (evenp ?current-argument)))
  (div (call-next-method) 2))
CLIPS> (+ 1 2)
3
CLIPS> (+ 4 6 4)
7
CLIPS>
```

It is important to emphasize that query and type restrictions on the wildcard parameter are applied to every argument grouped in the wildcard. Thus in the following example, the **>** and **length$** functions are actually called three times, since there are three arguments:

Example

```
CLIPS> (defmethod foo (($?any (> (length$ ?any) 2))) yes)
CLIPS> (foo 1 red 3)
yes
CLIPS>
```

In addition, a query restriction will never be examined if there are no arguments in the wildcard parameter range. For example, the the previous method*would* be applicable to a call to the generic function with no arguments because the query restriction is never evaluated:

Example

```
CLIPS> (foo)
yes
CLIPS>
```

Typically query restrictions applied to the entire wildcard parameter are testing the cardinality (the number of arguments passed to the method). In cases like this where the type is irrelevant to the test, the query restriction can be attached to a regular parameter instead to improve performance. Thus the previous method could be improved as follows:

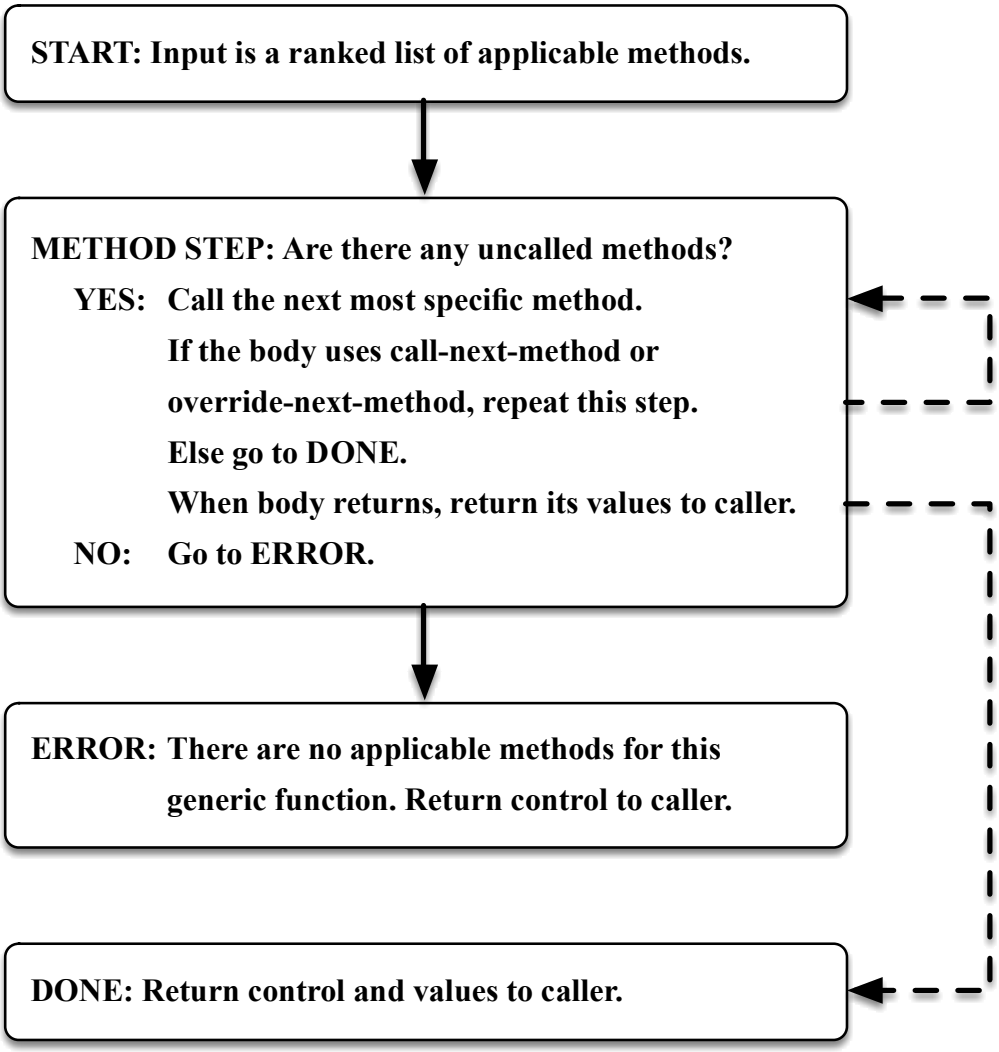
Example

```
CLIPS> (clear)
CLIPS> (defmethod foo ((?arg (> (length$ ?any) 1)) $?any) yes)
CLIPS> (foo)
[GENRCEXE1] No applicable methods for foo.
FALSE
CLIPS>
```

This approach should not be used if the types of the arguments grouped by the wildcard must be verified prior to safely evaluating the query restriction.

8.5 Generic Dispatch

When a generic function is called, CLIPS selects the method for that generic function with highest precedence for which parameter restrictions are satisfied by the arguments. This method is executed, and its value is returned as the value of the generic function. This entire process is referred to as the **generic dispatch**. Below is a flow diagram summary:



The solid arrows indicate automatic control transfer by the generic dispatch. The dashed arrows indicate control transfer that can only be accomplished by the use or lack of the use of call-next-method or override-next-method.

8.5.1 Applicability of Methods Summary

An explicit (user-defined) method is applicable to a generic function call if the following three conditions are met: 1) its name matches that of the generic function, 2) it accepts at least as many arguments as were passed to the generic function, and 3) every argument of the generic function satisfies the corresponding parameter restriction (if any) of the method.

Method restrictions are examined from left to right. As soon as one restriction is not satisfied, the method is abandoned, and the rest of the restrictions (if any) are not examined.

When a standard CLIPS system function is overloaded, CLIPS forms an implicit method definition corresponding to that system function. This implicit method is derived from the argument restriction string for the external **DefineFunction2** call defining that function to CLIPS (see the *Advanced Programming Guide*). This string can be accessed with the function **get-function-restrictions**. The specification of this implicit method can be examined with the **list-defmethods** or **get-method-restrictions** functions. The method that CLIPS will form for a system function can be derived by the user from the BNF given in this document. For example,

```
(+ <number> <number>+)
```

would yield the following method for the '+' function:

```
(defmethod + ((?first NUMBER) (?second NUMBER) ($?rest NUMBER))
   ...)
```

The method definition is used to determine the applicability and precedence of the system function to the generic function call.

The following system functions cannot be overloaded, and CLIPS will generate an error if an attempt is made to do so.

active-duplicate-instance
active-initialize-instance
active-make-instance
active-message-duplicate-instance
active-message-modify-instance
active-modify-instance
any-instancep
assert
bind
break
call-next-handler
call-next-method
call-specific-method
delayed-do-for-all-facts
delayed-do-for-all-instances
do-for-all-facts
do-for-all-instances
do-for-fact
do-for-instance
duplicate
duplicate-instance
expand$
find-all-facts
find-all-instances
find-fact
find-instance
foreach
if
make-instance
initialize-instance
loop-for-count
message-duplicate-instance
message-modify-instance
modify
modify-instance
next-handlerp
next-methodp
object-pattern-match-delay
override-next-handler
override-next-method
progn
progn$
return
switch
while

8.5.2 Method Precedence

When two or more methods are applicable to a particular generic function call, CLIPS must pick the one with highest **precedence** for execution. Method precedence is determined when a method is defined; the **list-defmethods** function can be used to examine the precedence of methods for a generic function.

The precedence between two methods is determined by comparing their parameter restrictions. In general, the method with the most specific parameter restrictions has the highest precedence. For example, a method that demands an integer for a particular argument will have higher precedence than a method which only demands a number. The exact rules of precedence between two methods are given in order below; the result of the first rule that establishes precedence is taken.

1) The parameter restrictions of both methods are positionally compared from left to right. In other words, the first parameter restriction in the first method is matched against the first parameter restriction in the second method, and so on. The comparisons between these pairs of parameter restrictions from the two methods determine the overall precedence between the two methods. The result of the first pair of parameter restrictions that specifies precedence is taken. The following rules are applied in order to a parameter pair; the result of the first rule that establishes precedence is taken.

   a) A regular parameter has precedence over a wildcard parameter.

   b) The most specific type restriction on a particular parameter has priority. A class is more specific than any of its superclasses.

   c) A parameter with a query restriction has priority over one that does not.

2) The method with the greater number of regular parameters has precedence.

3) A method without a wildcard parameter has precedence over one that does

4) A method defined before another one has priority.

If there are multiple classes on a single restriction, determining specificity is slightly more complicated. Since all precedence determination is done when the new method is defined, and the actual class of the generic function argument will not be known until run-time, arbitrary (but deterministic) rules are needed for determining the precedence between two class lists. The two class lists are examined by pairs from left to right, e.g. the pair of first classes from both lists, the pair of second classes from both lists and so on. The first pair containing a class and its superclass specify precedence. The class list containing the subclass has priority. If no class pairs specify precedence, then the shorter class list has priority. Otherwise, the class lists do not specify precedence between the parameter restrictions.

Example 1

```
; The system operator '+' is an implicit method       ; #1
; Its definition provided by the system is:
; (defmethod + ((?a NUMBER) (?b NUMBER) ($?rest NUMBER)))

(defmethod + ((?a NUMBER) (?b INTEGER)))              ; #2

(defmethod + ((?a INTEGER) (?b INTEGER)))             ; #3

(defmethod + ((?a INTEGER) (?b NUMBER)))              ; #4

(defmethod + ((?a NUMBER) (?b NUMBER)                 ; #5
              ($?rest PRIMITIVE)))

(defmethod + ((?a NUMBER)                             ; #6
              (?b INTEGER (> ?b 2))))

(defmethod + ((?a INTEGER (> ?a 2))                   ; #7
              (?b INTEGER (> ?b 3))))

(defmethod + ((?a INTEGER (> ?a 2))                   ; #8
              (?b NUMBER)))
```

The precedence would be: #7,#8,#3,#4,#6,#2,#1,#5. The methods can be immediately partitioned into three groups of decreasing precedence according to their restrictions on the first parameter: A) methods which have a query restriction and a type restriction of INTEGER (#7,#8), B) methods which have a type restriction of INTEGER (#3,#4), and C) methods which have a type restriction of NUMBER (#1,#2,#5,#6). Group A has precedence over group B because parameters with query restrictions have priority over those that do not. Group B has precedence over group C because INTEGER is a subclass of NUMBER. Thus, the ordering so far is: (#7,#8),(#3,#4),(#1,#2,#5,#6). Ordering between the methods in a particular set of parentheses is not yet established.

The next step in determining precedence between these methods considers their restrictions on the second parameter. #7 has priority over #8 because INTEGER is a subclass of NUMBER. #3 has priority over #4 because INTEGER is a subclass of NUMBER. #6 and #2 have priority over #1 and #5 because INTEGER is a subclass of NUMBER. #6 has priority over #2 because it has a query restriction and #2 does not. Thus the ordering is now: #7,#8,#3,#4,#6,#2,(#1,#5).

The restriction on the wildcard argument yields that #1 (the system function implicit method) has priority over #5 since NUMBER is a sublclass of PRIMITIVE. This gives the final ordering: #7,#8,#3,#4,#6,#2,#1,#5.

Example 2

```
(defmethod foo ((?a NUMBER STRING)))         ; #1

(defmethod foo ((?a INTEGER LEXEME)))        ; #2
```

The precedence would be #2,#1. Although STRING is a subclass of LEXEME, the ordering is still #2,#1 because INTEGER is a subclass of NUMBER, and NUMBER/INTEGER is the leftmost pair in the class lists.

Example 3

```
(defmethod foo ((?a MULTIFIELD STRING)))     ; #1

(defmethod foo ((?a LEXEME)))                ; #2
```

The precedence would be #2,#1 because the classes of the first pair in the type restriction (MULTIFIELD/LEXEME) are unrelated and #2 has fewer classes in its class list.

Example 4

```
(defmethod foo ((?a INTEGER LEXEME)))        ; #1

(defmethod foo ((?a STRING NUMBER)))         ; #2
```

Both pairs of classes (INTEGER/STRING and LEXEME/NUMBER) are unrelated, and the class lists are of equal length. Thus, the precedence is taken from the order of definition: #1,#2.

8.5.3 Shadowed Methods

If one method must be called by another method in order to be executed, the first function or method is a said to be **shadowed** by the second method. Normally, only one method or system function will be applicable to a particular generic function call. If there is more than one applicable method, the generic dispatch will only execute the one with highest precedence. Letting the generic dispatch automatically handle the methods in this manner is called the **declarative** technique, for the declarations of the method restrictions dictate which method gets executed in specific circumstances. However, the functions **call-next-method** and **override-next-method** may also be used which allow a method to execute the method that it is shadowing. This is called the **imperative** technique, since the method execution itself plays a role in the generic dispatch. This is *not* recommended unless it is absolutely necessary. In most circumstances, only one piece of code should need to be executed for a particular set of arguments. Another imperative technique is to use the function **call-specific-method** to override method precedence.

8.5.4 Method Execution Errors

If an error occurs while any method for a generic function call is executing, any actions in the current method not yet executed will be aborted, any methods not yet called will be aborted, and the generic function will return the symbol FALSE. The lack of any applicable methods for a set of generic function arguments is considered a method execution error.

8.5.5 Generic Function Return Value

The return value of a generic function is the return value of the applicable method with the highest precedence. Each applicable method that is executed can choose to ignore or capture the return value of any method that it is shadowing.

The return value of a particular method is the last action evaluated by that method.

Section 9:
CLIPS Object Oriented Language

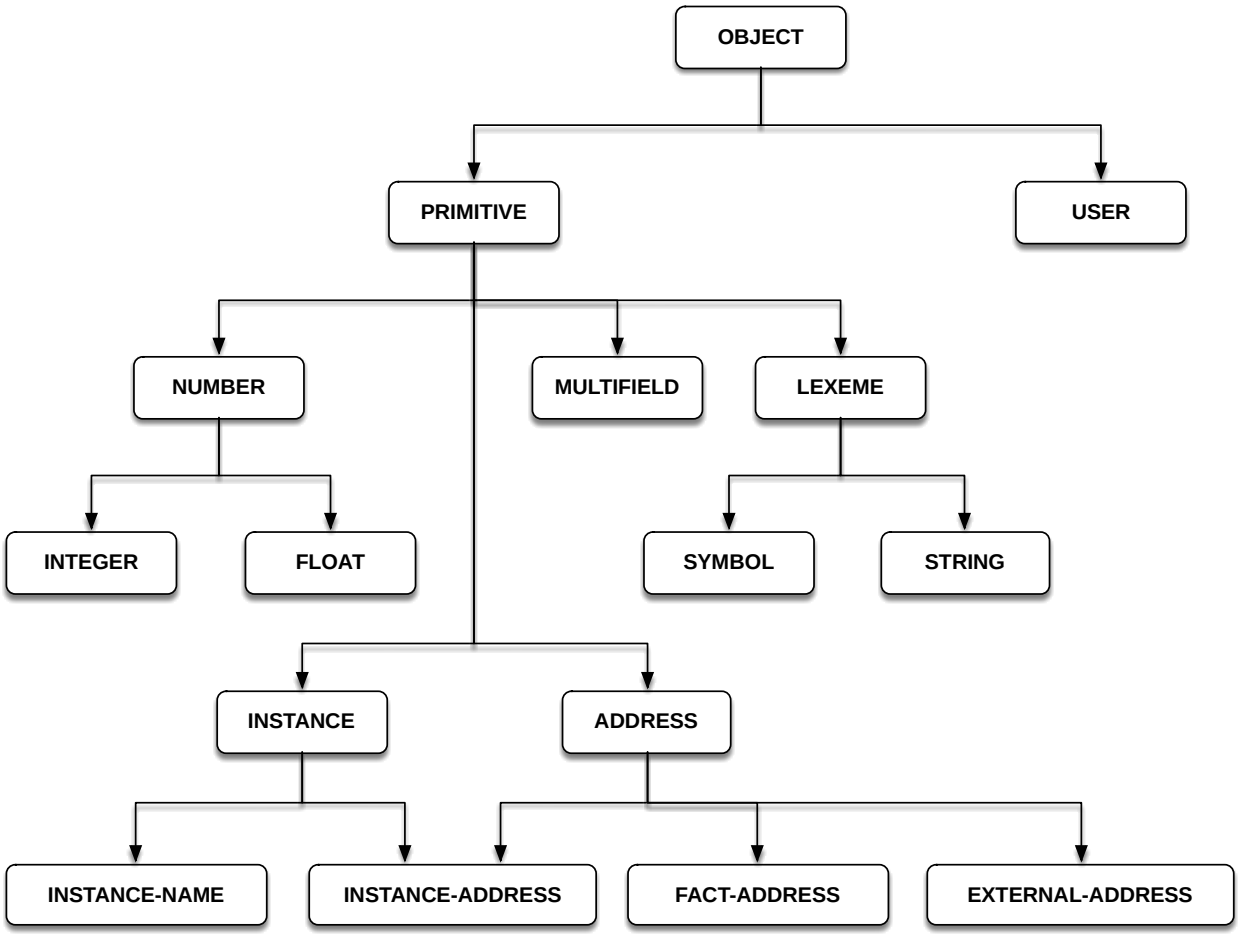
This section provides the comprehensive details of the CLIPS Object-Oriented Language (COOL). Sections 2.3.1, 2.4.2 and 2.5.2.3 explain object references and structure. Section 2.6 gives a high level overview of COOL. This section assumes a complete understanding of the material given in the listed sections.

9.1 Background

COOL is a hybrid of features from many different OOP systems as well as new ideas. For example, object encapsulation concepts are similar to those in Smalltalk, and the Common Lisp Object System (CLOS) provides the basis for multiple inheritance rules. A mixture of ideas from Smalltalk, CLOS and other systems form the foundation of messages. Section 8.1 explains an important contrast between the terms **method** and **message-handler** in CLIPS.

9.2 Predefined System Classes

COOL provides sixteen system classes: OBJECT, USER, PRIMITIVE, NUMBER, INTEGER, FLOAT, INSTANCE, INSTANCE-NAME, INSTANCE-ADDRESS, ADDRESS, FACT-ADDRESS, EXTERNAL-ADDRESS, MULTIFIELD, LEXEME, SYMBOL and STRING. The user may not delete or modify any of these classes. The diagram illustrates the inheritance relationships between these classes.



All of these system classes are **abstract** classes, which means that their only use is for inheritance (**direct** instances of this class are illegal). None of these classes have slots, and, except for the class USER, none of them have message-handlers. However, the user may explicitly attach message-handlers to all of the system classes except for INSTANCE, INSTANCE-ADDRESS and INSTANCE-NAME. The OBJECT class is a superclass of all other classes, including user-defined classes. All user-defined classes should (but are not required to) inherit directly or indirectly from the class USER, since this class has all of the standard system message-handlers, such as initialization and deletion, attached to it. Section 9.4 describes these system message-handlers.

The PRIMITIVE system class and all of its subclasses are provided mostly for use in generic function method restrictions, but message-handlers and new subclasses may be attached if desired. However, the three primitive system classes INSTANCE, INSTANCE-ADDRESS and INSTANCE-NAME are provided strictly for use in methods (particularly in forming implicit methods for overloaded system functions - see section 8.5.1) and as such cannot have subclasses or message-handlers attached to them.

9.3 Defclass Construct

A **defclass** is a construct for specifying the properties (slots) and behavior (message-handlers) of a class of objects. A defclass consists of five elements: 1) a name, 2) a list of superclasses from which the new class inherits slots and message-handlers, 3) a specifier saying whether or not the creation of direct instances of the new class is allowed, 4) a specifier saying whether or not instances of this class can match object patterns on the LHS of rules and 5) a list of slots specific to the new class. All user-defined classes must inherit from at least one class, and to this end COOL provides predefined system classes for use as a base in the derivation of new classes.

Any slots explicitly given in the defclass override those gotten from inheritance. COOL applies rules to the list of superclasses to generate a class precedence list (see section 9.3.1) for the new class. Facets (see section 9.3.3) further describe slots. Some examples of facets include: default value, cardinality, and types of access allowed.

Syntax

Defaults are in ***bold italics***.

```
(defclass <name> [<comment>]
  (is-a <superclass-name>+)
  [<role>]
  [<pattern-match-role>]
  <slot>*
  <handler-documentation>*)

<role>  ::= (role concrete | abstract)

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

<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
```

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 conflicting definitions of another class found later in the list. A class that comes before another class in the list is said to be more **specific**. All class precedence lists will terminate in the system class OBJECT, and most (if not all) class precedence lists for user-defined classes will terminate in the system classes USER and OBJECT. The class precedence list can be listed using the **describe-class** function.

9.3.1.1 Multiple Inheritance Rules

COOL uses the inheritance hierarchy of the direct superclasses to determine the class precedence list for a new class. COOL recursively applies the following two rules to the direct superclasses:

1) A class has higher precedence than any of its superclasses.

2) A class specifies the precedence between its direct superclasses.

If more than one class precedence list would satisfy these rules, COOL chooses the one most similar to a strict preorder depth-first traversal. This heuristic attempts to preserve “family trees” to the greatest extent possible. For example, if a child inherited genetic traits from a mother and father, and the mother and father each inherited traits from their parents, the child’s class precedence list would be: child mother maternal-grandmother maternal-grandfather father

paternal-grandmother paternal-grandfather. There are other orderings which would satisfy the rules (such as child mother father paternal-grandfather maternal-grandmother paternal-grandmother maternal-grandfather), but COOL chooses the one which keeps the family trees together as much as possible.

Example 1

```
(defclass A (is-a USER))
```

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

Example 2

```
(defclass B (is-a USER))
```

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

Example 3

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

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

Example 4

```
(defclass D (is-a B A))
```

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.

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 classes specified in the *is-a* attribute effects the behavior of the class, you should consider whether your program design is needlessly complex.

9.3.2 Class Specifiers

9.3.2.1 Abstract and Concrete Classes

An **abstract** class is intended for inheritance only, and no direct instances of this class can be created. A **concrete** class can have direct instances. Using the abstract role specifier in a defclass will cause COOL to generate an error if **make-instance** is ever called for this class. If the abstract or concrete descriptor for a class is not specified, it is determined by inheritance. For the purpose of role inheritance, system defined classes behave as concrete classes. Thus a class which inherits from USER will be concrete if no role is specified.

9.3.2.2 Reactive and Non-Reactive Classes

Objects of a **reactive** class can match object patterns in a rule. Objects of a **non-reactive** class cannot match object patterns in a rule and are not considered when the list of applicable classes are determined for an object pattern. An **abstract** class cannot be **reactive**. If the reactive or non-reactive descriptor for a class is not specified, it is determined by inheritance. For the purpose of pattern-match inheritance, system defined classes behave as reactive classes unless the inheriting class is **abstract**.

9.3.3 Slots

Slots are placeholders for values associated with instances of a user-defined class. Each instance has a copy of the set of slots specified by the immediate class as well as any obtained from inheritance. Only available memory limits the number of slots. The name of a slot may be any symbol with the exception of the keywords *is-a* and *name* which are reserved for use in object patterns.

To determine the set of slots for an instance, the class precedence list for the instance's class is examined in order from most specific to most general (left to right). A class is more specific than its superclasses. Slots specified in any of the classes in the class precedence list are given to the instance, with the exception of no-inherit slots (see section 9.3.3.5). If a slot is inherited from more than one class, the definition given by the more specific class takes precedence, with the exception of composite slots (see section 9.3.3.6).

Example

```
(defclass A (is-a USER)
   (slot fooA)
   (slot barA))

(defclass B (is-a A)
   (slot fooB)
   (slot barB))
```

The class precedence list of A is: A USER OBJECT. Instances of A will have two slots: fooA and barA. The class precedence list of B is: B A USER OBJECT. Instances of B will have four slots: fooB, barB, fooA and barA.

Just as slots make up classes, **facets** make up slots. Facets describe various features of a slot that hold true for all objects which have the slot: default value, storage, access, inheritance propagation, source of other facets, pattern-matching reactivity, visibility to subclass message-handlers, the automatic creation of message-handlers to access the slot, the name of the message to send to set the slot and constraint information. Each object can still have its own value for a slot, with the exception of shared slots (see section 9.3.3.3).

9.3.3.1 Slot Field Type

A slot can hold either a single-field or multifield value. By default, a slot is single-field. The keyword **multislot** specifies that a slot can hold a multifield value comprised of zero or more fields, and the keywords **slot** or **single-slot** specify that the slot can hold one value. Multifield slot values are stored as multifield values and can be manipulated with the standard multifield functions, such as **nth$** and **length$**, once they are retrieved via messages. COOL also provides functions for setting multifield slots, such as **slot-insert$**. Single-field slots are stored as a CLIPS primitive type, such as integer or string.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (multislot foo
      (default abc def ghi)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (nth$ 2 (send [a] get-foo))
def
CLIPS>
```

9.3.3.2 Default Value Facet

The **default** and **default-dynamic** facets can be used to specify an initial value given to a slot when an instance of the class is created or initialized. By default, a slot will have a default value that is derived from the slot's constraint facets (see sections 9.3.3.11 and 11.5). Default values are directly assigned to slots without the use of messages, unlike slot overrides in a **make-instance** call (see section 9.6.1).

The **default** facet is a static default: the specified expression is evaluated once when the class is defined, and the result is stored with the class. This result is assigned to the appropriate slot when a new instance is created. If the keyword ?DERIVE is used for the default value, then a default value is derived from the constraints for the slot (see section 11.5 for more details). By default, the default attribute for a slot is (default ?DERIVE). If the keyword ?NONE is used for the default value, then the slot is not assigned a default value. Using this keyword causes make-instance to require a slot-override for that slot when an instance is created.

The **default-dynamic** facet is a dynamic default: the specified expression is evaluated every time an instance is created, and the result is assigned to the appropriate slot.

Example

```
CLIPS> (clear)
CLIPS> (setgen 1)
1
CLIPS>
```

```
(defclass A (is-a USER)
   (slot foo (default-dynamic (gensym))))
CLIPS> (make-instance a1 of A)
[a1]
CLIPS> (make-instance a2 of A)
[a2]
CLIPS> (send [a1] get-foo)
gen1
CLIPS> (send [a2] get-foo)
gen2
CLIPS>
```

9.3.3.3 Storage Facet

The actual value of an instance's copy of a slot can either be stored with the instance or with the class. The **local** facet specifies that the value be stored with the instance, and this is the default. The **shared** facet specifies that the value be stored with the class. If the slot value is locally stored, then each instance can have a separate value for the slot. However, if the slot value is stored with the class, all instances will have the same value for the slot. Anytime the value is changed for a shared slot, it will be changed for all instances with that slot.

A shared slot will always pick up a dynamic default value from a defclass when an instance is created or initialized, but the shared slot will ignore a static default value unless it does not currently have a value. Any changes to a shared slot will cause pattern-matching for rules to be updated for all reactive instances containing that slot.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot foo (storage shared)
             (default 1))
   (slot bar (storage shared)
             (default-dynamic 2))
   (slot woz (storage local)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print)
[a] of A
(foo 1)
(bar 2)
(woz nil)
CLIPS> (send [a] put-foo 56)
56
CLIPS> (send [a] put-bar 104)
104
CLIPS> (make-instance b of A)
[b]
CLIPS> (send [b] print)
[b] of A
(foo 56)
(bar 2)
(woz nil)
CLIPS> (send [b] put-foo 34)
```

```
34
CLIPS> (send [b] put-woz 68)
68
CLIPS> (send [a] print)
[a] of A
(foo 34)
(bar 2)
(woz nil)
CLIPS> (send [b] print)
[b] of A
(foo 34)
(bar 2)
(woz 68)
CLIPS>
```

9.3.3.4 Access Facet

There are three types of access facets which can be specified for a slot: **read-write**, **read-only**, and **initialize-only**. The **read-write** facet is the default and says that a slot can be both written 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)
   (slot foo (access read-write))
   (slot bar (access read-only) (default abc))
   (slot woz (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>
```

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)
   (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 that 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.

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
Direct Subclasses:
-------------------------------------------------------------------------------
SLOTS : FLD DEF PRP ACC STO MCH SRC VIS CRT OVRD-MSG SOURCE(S)
foo   : MLT STC INH  R  SHR RCT CMP PRV  R  NIL      A B

Constraint information for slots:

SLOTS : SYM STR INN INA EXA FTA INT FLT
foo   :  +   +   +   +   +   +   +   +  RNG:[-oo..+oo] CRD:[0..+oo]
-------------------------------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
get-foo primary in class A
get-foo primary in class B
*******************************************************************************
===============================================================================
CLIPS>
```

9.3.3.7 Pattern-Match Reactivity Facet

Normally, any change to a slot of an instance will be considered as a change to the instance for purposes of pattern-matching. However, it is possible to indicate that changes to a slot of an instance should not cause pattern-matching. The **reactive** facet specifies that changes to a slot trigger pattern-matching, and this is the default. The **non-reactive** facet specifies that changes to a slot do not affect pattern-matching.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot foo (pattern-match non-reactive)))
CLIPS>
(defclass B (is-a USER)
   (slot foo))
CLIPS>
(defrule Create
  ?ins<-(object (is-a A | B))
=>
```

```
  (println "Create " (instance-name ?ins)))
CLIPS>
(defrule Foo-Access
  ?ins<-(object (is-a A | B) (foo ?))
=>
  (println "Foo-Access " (instance-name ?ins)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (make-instance b of B)
[b]
CLIPS> (run)
Create [b]
Foo-Access [b]
Create [a]
CLIPS> (send [a] put-foo 1)
1
CLIPS> (send [b] put-foo 1)
1
CLIPS> (run)
Foo-Access [b]
CLIPS>
```

9.3.3.8 Visibility Facet

Normally, only message-handlers attached to the class in which a slot is defined may directly access the slot. However, it is possible to allow message-handlers attached to superclasses or subclasses which inherit the slot to directly access the slot as well. Declaring the **visibility** facet to be **private** specifies that only the message-handlers of the defining class may directly access the slot, and this is the default. Declaring the **visibility** facet to be **public** specifies that the message-handlers and subclasses that inherit the slot and superclasses may also directly access the slot.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot foo (visibility private)))
CLIPS>
(defclass B (is-a A))
CLIPS>
(defmessage-handler B get-foo ()
  ?self:foo)
[MSGFUN6] Private slot foo of class A cannot be accessed directly by handlers
attached to class B

[PRCCODE3] Undefined variable self:foo referenced in message-handler.

ERROR:
(defmessage-handler MAIN::B get-foo
   ()
   ?self:foo
   )
CLIPS>
```

9.3.3.9 Create-Accessor Facet

The **create-accessor** facet instructs CLIPS to automatically create *explicit* message-handlers for reading and/or writing a slot. By default, implicit slot-accessor message-handlers are created for every slot. While these message-handlers are real message-handlers and can be manipulated as such, they have no pretty-print form and cannot be directly modified by the user.

If the value **?NONE** is specified for the facet, no message-handlers are created.

If the value **read** is specified for the facet, CLIPS creates the following message-handler:

```
(defmessage-handler <class> get-<slot-name> primary ()
   ?self:<slot-name>)
```

If the value **write** is specified for the facet, CLIPS creates the following message-handler for single-field slots:

```
(defmessage-handler <class> put-<slot-name> primary (?value)
   (bind ?self:<slot-name> ?value)
```

or the following message-handler for multifield slots:

```
(defmessage-handler <class> put-<slot-name> primary ($?value)
   (bind ?self:<slot-name> ?value)
```

If the value **read-write** is specified for the facet, both the get- and one of the put- message-handlers are created.

If accessors are required that do not use static slot references (see sections 9.4.2, 9.6.3 and 9.6.4), then user must define them explicitly with the defmessage-handler construct.

The **access** facet affects the default value for the **create-accessor** facet. If the **access** facet is **read-write**, then the default value for the **create-accessor** facet is **read-write**. If the **access** facet is **read-only**, then the default value is **read**. If the access facet is **initialize-only**, then the default is **?NONE**.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot foo (create-accessor write))
  (slot bar (create-accessor read))
CLIPS> (make-instance a of A (foo 36))
[a]
CLIPS> (make-instance b of A (bar 45))
[MSGFUN1] No applicable primary message-handlers found for put-bar.
FALSE
CLIPS>
```

9.3.3.10 Override-Message Facet

There are several COOL support functions that set slots via use of message-passing, e.g., **make-instance**, **initialize-instance**, **message-modify-instance** and **message-duplicate-instance**. By default, all these functions attempt to set a slot with the message called **put-<slot-name>**. However, if the user has elected not to use standard slot-accessors and wishes these functions to be able to perform slot-overrides, then the **override-message** facet can be used to indicate what message to send instead.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot special (override-message special-put)))
CLIPS>
(defmessage-handler A special-put primary (?value)
  (bind ?self:special ?value))
CLIPS> (watch messages)
CLIPS> (make-instance a of A (special 65))
MSG >> create ED:1 (<Instance-a>)
MSG << create ED:1 (<Instance-a>)
MSG >> special-put ED:1 (<Instance-a> 65)
MSG << special-put ED:1 (<Instance-a> 65)
MSG >> init ED:1 (<Instance-a>)
MSG << init ED:1 (<Instance-a>)
[a]
CLIPS> (unwatch messages)
CLIPS>
```

9.3.3.11 Constraint Facets

The syntax and functionality of single and multifield constraint facets (attributes) are described in detail in Section 11. Static and dynamic constraint checking for classes and their instances is supported. Static checking is performed when constructs or commands that specify slot information are being parsed. Object patterns used on the LHS of a rule are also checked to determine if constraint conflicts exist among variables used in more that one slot. Errors for inappropriate values are immediately signaled. Static checking is enabled by default. This behavior can be changed using the **set-static-constraint-checking** function. Dynamic checking is also supported. If dynamic checking is enabled, then new instances have their values checked whenever they are set (e.g. initialization, slot-overrides, and put- access). This dynamic checking is disabled by default. This behavior can be changed using the **set-dynamic-constraint-checking** function. If an violation occurs when dynamic checking is being performed, then execution will be halted.

Regardless of whether static or dynamic checking is enabled, multifield values can never be stored in single-field slots. Single-field values are converted to a multifield value of length one when storing in a multifield slot. In addition, the evaluation of a function that has no return value is always illegal as a slot value.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (multislot foo (type SYMBOL)
                  (cardinality 2 3)))
CLIPS> (make-instance a of A (foo 45))
[a]
CLIPS> (set-dynamic-constraint-checking TRUE)
FALSE
CLIPS> (make-instance a of A (foo red 5.0))
[CSTRNCHK1] (red 5.0) for slot foo of instance [a] found in put-foo primary in
class A does not match the allowed types.
[PRCCODE4] Execution halted during the actions of message-handler put-foo primary
in class A
FALSE
CLIPS> (make-instance a of A (foo red))
[CSTRNCHK1] (red) for slot foo of instance [a] found in put-foo primary in class A
does not satisfy the cardinality restrictions.
[PRCCODE4] Execution halted during the actions of message-handler put-foo primary
in class A
FALSE
CLIPS>
```

9.3.4 Message-handler Documentation

COOL allows the user to forward declare the message-handlers for a class within the defclass statement. These declarations are for documentation only and are ignored by CLIPS. The defmessage-handler construct must be used to actually add message-handlers to a class. Message-handlers can later be added which are not documented in the defclass.

Example

```
CLIPS> (clear)
CLIPS>
(defclass rectangle (is-a USER)
   (slot side-a (default 1))
   (slot side-b (default 1))
   (message-handler find-area))
CLIPS>
(defmessage-handler rectangle find-area ()
   (* ?self:side-a ?self:side-b))
CLIPS>
(defmessage-handler rectangle print-area ()
   (println (send ?self find-area)))
CLIPS>
```

9.4 Defmessage-handler Construct

Objects are manipulated by sending them messages via the function **send**. The result of a message is a useful return-value or side-effect. A **defmessage-handler** is a construct for specifying the behavior of a class of objects in response to a particular message. The implementation of a message is made up of pieces of procedural code called message-handlers

(or handlers for short). Each class in the class precedence list of an object's class can have handlers for a message. In this way, the object's class and all its superclasses share the labor of handling the message. Each class's handlers handle the part of the message that is appropriate to that class. Within a class, the handlers for a particular message can be further subdivided into four types or categories: **primary**, **before**, **after** and **around**. The intended purposes of each type are summarized in the chart below:

Type	Role for the Class
primary	Performs the majority of the work for the message
before	Does auxiliary work for a message before the primary handler executes
after	Does auxiliary work for a message after the primary handler executes
around	Sets up an environment for the execution of the rest of the handlers

Before and after handlers are for side-effects only; their return values are always ignored. Before handlers execute before the primary ones, and after message-handlers execute after the primary ones. The return value of a message is generally given by the primary message-handlers, but around handlers can also return a value. Around message-handlers allow the user to wrap code around the rest of the handlers. They begin execution before the other handlers and pick up again after all the other message-handlers have finished.

A primary handler provides the part of the message implementation which is most specific to an object, and thus the primary handler attached to the class closest to the immediate class of the object overrides other primary handlers. Before and after handlers provide the ability to pick up behavior from classes that are more general than the immediate class of the object, thus the message implementation uses all handlers of this type from all the classes of an object. When only the roles of the handlers specify which handlers get executed and in what order, the message is said to be **declaratively** implemented. However, some message implementations may not fit this model well. For example, the results of more than one primary handler may be needed. In cases like this, the handlers themselves must take part in deciding which handlers get executed and in what order. This is called the **imperative** technique. Around handlers provide imperative control over all other types of handlers except more specific around handlers. Around handlers can change the environment in which other handlers execute and modify the return value for the entire message. A message implementation should use the declarative technique if at all possible because this allows the handlers to be more independent and modular.

A defmessage-handler is comprised of seven elements: 1) a class name to which to attach the handler (the class must have been previously defined), 2) a message name to which the handler will respond, 3) an optional type (the default is primary), 4) an optional comment, 5) a list of parameters that will be passed to the handler during execution, 6) an optional wildcard parameter and 7) a series of expressions which are executed in order when the handler is called. The return-value of a message-handler is the evaluation of the last expression in the body.

Syntax

Defaults are in ***bold italics***.

```
(defmessage-handler <class-name> <message-name>
                    [<handler-type>] [<comment>]
   (<parameter>* [<wildcard-parameter>])
   <action>*)

<handler-type>         ::= around | before | primary | after
<parameter>            ::= <single-field-variable>
<wildcard-parameter>   ::= <multifield-variable>
```

Message-handlers are uniquely identified by class, name and type. Message-handlers are never called directly. When the user sends a message to an object, CLIPS selects and orders the applicable message-handlers attached to the object's class(es) and then executes them. This process is termed the **message dispatch**.

Example

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
CLIPS>
(defmessage-handler A delete before ()
    (println "Deleting an instance of the class A..."))
CLIPS>
(defmessage-handler USER delete after ()
   (println "System completed deletion of an instance."))
CLIPS> (watch instances)
CLIPS> (make-instance a of A)
==> instance [a] of A
[a]
CLIPS> (send [a] delete)
Deleting an instance of the class A...
<== instance [a] of A
System completed deletion of an instance.
TRUE
CLIPS> (unwatch instances)
CLIPS>
```

9.4.1 Message-handler Parameters

A message-handler may accept *exactly* or *at least* a specified number of arguments, depending on whether a wildcard parameter is used or not. The regular parameters specify the minimum number of arguments that must be passed to the handler. Each of these parameters may be referenced like a normal single-field variable within the actions of the handler. If a wildcard parameter is present, the handler may be passed any number of arguments greater than or equal to the minimum. If no wildcard parameter is present, then the handler must be passed exactly the number of arguments specified by the regular parameters. All arguments to a handler that do not correspond to a regular parameter are grouped into a multifield value that can be referenced by the wildcard parameter. The standard CLIPS multifield functions, such as **length$** and **expand$**, can be applied to the wildcard parameter.

Handler parameters have no bearing on the applicability of a handler to a particular message (see section 9.5.1). However, if the number of arguments is inappropriate, a message execution error (see section 9.5.4) will be generated when the handler is called. Thus, the number of arguments accepted should be consistent for all message-handlers applicable to a particular message.

Example

```
CLIPS> (clear)
CLIPS>
(defclass CAR (is-a USER)
   (slot front-seat)
   (multislot trunk)
   (slot trunk-count))
CLIPS>
(defmessage-handler CAR put-items-in-car (?item $?rest)
   (bind ?self:front-seat ?item)
   (bind ?self:trunk ?rest)
   (bind ?self:trunk-count (length$ ?rest)))
CLIPS> (make-instance Pinto of CAR)
[Pinto]
CLIPS> (send [Pinto] put-items-in-car bag-of-groceries
        tire suitcase)
2
CLIPS> (send [Pinto] print)
[Pinto] of CAR
(front-seat bag-of-groceries)
(trunk tire suitcase)
(trunk-count 2)
CLIPS>
```

9.4.1.1 Active Instance Parameter

The term **active instance** refers to an instance that is responding to a message. All message-handlers have an implicit parameter called **?self** which binds the active instance for a message. This parameter name is reserved and cannot be explicitly listed in the message-handler's parameters, nor can it be rebound within the body of a message-handler.

Example

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
CLIPS> (make-instance a of A)
[a]
CLIPS>
(defmessage-handler A print-args (?a ?b $?c)
   (println (instance-name ?self) " " ?a " " ?b
          " and " (length$ ?c) " extras: " ?c))
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 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.

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)
  (slot foo (default 1))
  (slot bar (default 2)))
CLIPS>
(defmessage-handler A print-all-slots ()
  (println ?self:foo " " ?self:bar))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print-all-slots)
1 2
CLIPS>
```

The **bind** function can also take advantage of this shorthand notation to set the value of a slot.

Syntax

```
(bind ?self:<slot-name> <value>*)
```

Example

```
CLIPS>
(defmessage-handler A set-foo (?value)
  (bind ?self:foo ?value))
CLIPS> (send [a] set-foo 34)
34
CLIPS>
```

Direct slot accesses are statically bound to the appropriate slot in the defclass when the message-handler is defined. Care must be taken when these direct slot accesses can be executed as the result of a message sent to an instance of a subclass of the class to which the message-handler is attached. If the subclass has redefined the slot, the direct slot access contained in the message-handler attached to the superclass will fail. That message-handler accesses the slot in the superclass, not the subclass.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot foo (create-accessor read)))
CLIPS>
(defclass B (is-a A)
   (slot foo (create-accessor ?NONE)))
CLIPS> (make-instance b of B)
[b]
CLIPS> (send [b] get-foo)
[MSGPASS3] Static reference to slot foo of class A does not apply to [b] of B
[PRCCODE4] Execution halted during the actions of message-handler get-foo primary
in class A
FALSE
CLIPS>
```

In order for direct slot accesses in a superclass message-handler to apply to new versions of the slot in subclasses, the dynamic-put and dynamic-get functions must be used. However, the subclass slot must have public visibility for this to work (see section 9.3.3.8).

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot foo (create-accessor ?NONE)))
CLIPS>
(defmessage-handler A get-foo ()
  (dynamic-get foo))
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))
CLIPS>
(defmessage-handler A init before ()
   (println "Initializing a new instance of class A..."))
CLIPS> (make-instance a of A)
Initializing a new instance of class A...
[a]
CLIPS>
```

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. 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)
  (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>
```

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 Corniche)
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>
```

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

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))
```

9.4.4.7 Duplicating an Instance using Messages

Syntax

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

This handler duplicates an instance using messages. Slot values from the original instance and slot overrides are copied using put- and get- messages. If the name of the new instance created matches a currently existing instance-name, then the currently existing instance is deleted using a delete message. After creation, the new instance is sent a create message and then an init message. The slot-override expressions are passed as an EXTERNAL_ADDRESS data object to the message-duplicate handler. This message is used by the functions **message-duplicate-instance** and **active-message-duplicate-instance**.

9.4.4.8 Instance Creation

Syntax

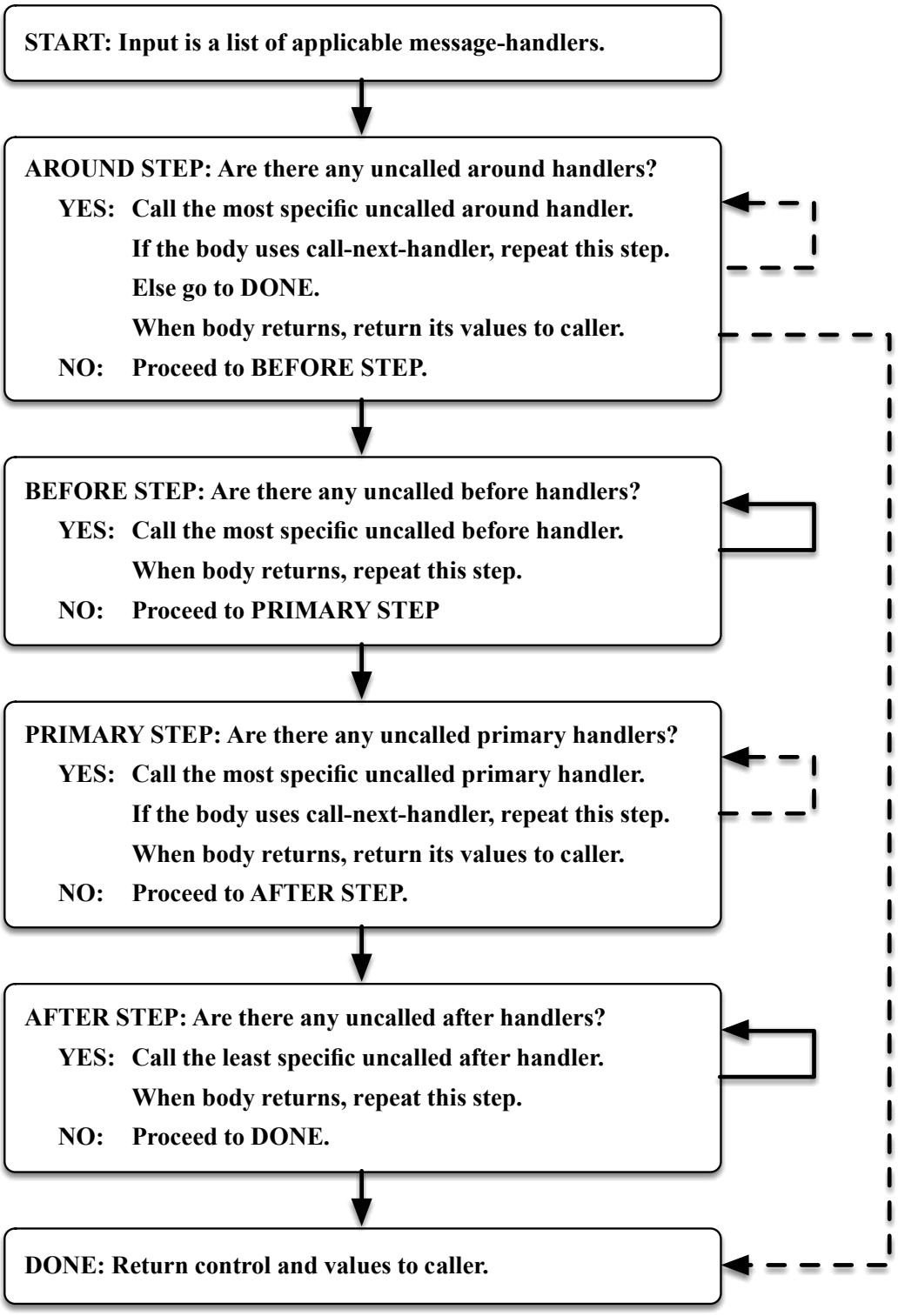
```
(defmessage-handler USER create primary ())
```

This handler is called after an instance is created, but before any slot initialization has occurred. The newly created instance is sent a **create** message. This handler performs no actions—It is provided so that instance creation can be detected by user-defined message-handlers. The

handler returns the symbol TRUE if the instance was successfully created, otherwise it returns the symbol FALSE.

9.5 Message Dispatch

When a message is sent to an object using the **send** function, CLIPS examines the class precedence list of the active instance's class to determine a complete set of message-handlers which are applicable to the message. CLIPS uses the roles (around, before, primary or after) and specificity of these message-handlers to establish an ordering and then executes them. A handler that is attached to a subclass of another message-handler's class is said to be more specific. This entire process is referred to as the **message dispatch**. Following is a flow diagram summary:



The solid arrows indicate automatic control transfer by the message dispatch system. The dashed arrows indicate control transfer that can only be accomplished by the use or lack of the use of **call-next-handler** (or **override-next-handler**).

9.5.1 Applicability of Message-handlers

A message-handler is applicable to a message if its name matches the message, and it is attached to a class which is in the class precedence list of the class of the instance receiving the message.

9.5.2 Message-handler Precedence

The set of all applicable message-handlers are sorted into four groups according to role, and these four groups are further sorted by class specificity. The around, before and primary handlers are ordered from most specific to most general, whereas after handlers are ordered from most general to most specific.

The order of execution is as follows: 1) around handlers begin execution from most specific to most general (each around handler must explicitly allow execution of other handlers), 2) before handlers execute (one after the other) from most specific to most general 3) primary handlers begin execution from most specific to most general (more specific primary handlers must explicitly allow execution of more general ones), 4) primary handlers finish execution from most general to most specific, 5) after handlers execute (one after the other) from most general to most specific and 6) around handlers finish execution from most general to most specific.

There must be at least one applicable primary handler for a message, or a message execution error will be generated.

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.

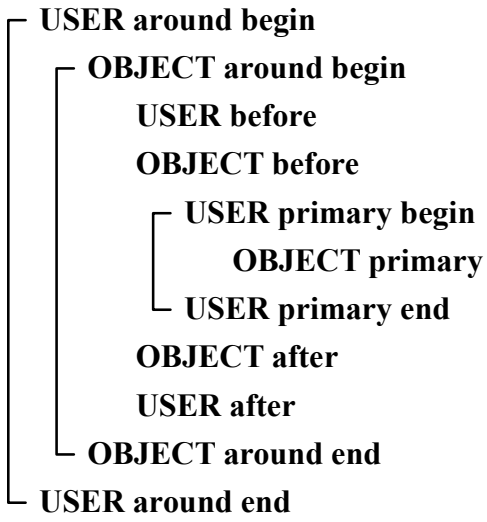
Messages should be implemented using the declarative technique, if possible. Only the handler roles will dictate which handlers get executed; only before and after handlers and the most specific primary handler are used. This allows each handler for a message to be completely independent of the other 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** allow a handler to execute the handler it is shadowing. A handler can call the same shadowed handler multiple times.

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 following diagram 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.

A lack of applicable of primary message-handlers and a handler being called with the wrong number of arguments are common message execution errors.

9.5.5 Message Return Value

The return value of call to the **send** function is the return value of the most specific around handler, or the most specific primary handler if there are no around handlers. The return value of a handler is the result of the evaluation of the last action in the handler.

The return values of the before and after handlers are ignored; they are for side-effects only. An around handler can choose to ignore or capture the return value of the next most specific around or primary handler. A primary handler can choose to ignore or capture the return value of a more general primary handler.

9.6 Manipulating Instances

Objects are manipulated by sending them messages. This is achieved by using the **send** function, which takes as arguments the destination object for the message, the message itself and any arguments which are to be passed to handlers.

Syntax

```
(send <object-expression>
  <message-name-expression> <expression>*)
```

Section 2.4.2 explains object references. The return value of **send** is the result of the message as explained in section 9.5.5.

The slots of an object may be read or set directly only within the body of a message-handler that is executing on behalf of a message that was sent to that object. This is how COOL implements the notion of encapsulation (see Section 2.6.2). Any action performed on an object by an external source, such as a rule or function, must be done with messages. There are two major exceptions: 1) objects which are not instances of user-defined classes (floating-point and integer numbers, symbols, strings, multifield values, fact-addresses and external-addresses) can be manipulated in the standard non-OOP manner of previous versions of CLIPS as well and 2) creation and initialization of an instance of a user-defined class are performed via the function **make-instance**.

9.6.1 Creating Instances

Like facts, instances of user-defined classes must be explicitly created by the user. Likewise, all instances are deleted during the **reset** command, and they can be loaded and saved similarly to facts. All operations involving instances require message-passing using the **send** function except for creation, since the object does not yet exist. A function called **make-instance** is used to create and initialize a new instance. This function implicitly sends first a create message and then an initialization message to the new object after allocation. The user can customize instance

initialization with daemons. **make-instance** also allows slot-overrides to change any predefined initialization for a particular instance. **make-instance** automatically delays all object pattern-matching activities for rules until all slot overrides have been processed. The function **active-make-instance** can be used if delayed pattern-matching is not desired. **active-make-instance** remembers the current state of delayed pattern-matching, explicitly turns delay on, and then restores it to its previous state once all slot overrides have been processed.

Syntax

```
(make-instance <instance-definition>)
(active-make-instance <instance-definition>)

<instance-definition> ::= [<instance-name-expression>] of
                                    <class-name-expression>
                                    <slot-override>*
<slot-override>       ::= (<slot-name-expression>
                                    <expression>*)
```

The return value of **make-instance** is the name of the new instance on success or the symbol FALSE on failure. The evaluation of <instance-name-expression> can either be an instance-name or a symbol. If <instance-name-expression> is not specified, then the function **gensym*** will be called to generate the instance-name.

The **make-instance** function performs the following steps in order:

1) If an instance of the specified name already exists, that instance receives a **delete** message, e.g. (send <instance-name> delete). If this fails for any reason, the new instance creation is aborted. Normally, the handler attached to class USER will respond to this message.

2) A new and uninitialized instance of the specified class is created with the specified name.

3) The new instance receives the **create** message, e.g. (send <instance-name> create). Normally, the handler attached to class USER will respond to this, although it performs no actions.

4) All slot-overrides are immediately evaluated and placed via **put-** messages, e.g. (send <instance-name> put-<slot-name> <expression>*). If there are any errors, the new instance is deleted.

5) The new instance receives the **init** message, e.g. (send <instance-name> init). Normally, the handler attached to class USER will respond to this message. This handler calls the **init-slots** function. This function uses defaults from the class definition (if any) for any slots which do not have slot-overrides. The class defaults are placed directly without the use of messages. If there are any errors, the new instance is deleted.

Example

```
CLIPS> (clear)
```

```
CLIPS>
(defclass A (is-a USER)
   (slot x (default 34))
   (slot y (default abc)))
CLIPS>
(defmessage-handler A put-x before (?value)
   (println "Slot x set with message."))
CLIPS>
(defmessage-handler A delete after ()
   (println "Old instance deleted."))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print)
[a] of A
(x 34)
(y abc)
CLIPS> (make-instance [a] of A (x 65))
Old instance deleted.
Slot x set with message.
[a]
CLIPS> (send [a] print)
a of A
(x 65)
(y abc)
CLIPS> (send [a] delete)
Old instance deleted.
TRUE
CLIPS>
```

9.6.1.1 Definstances Construct

Similar to deffacts, the **definstances** construct allows the specification of instances which will be created every time the **reset** command is executed. On every reset all current instances receive a **delete** message, and the equivalent of a **make-instance** function call is made for every instance specified in definstances constructs.

Syntax

```
(definstances <definstances-name> [active] [<comment>]
   <instance-template>*)
<instance-template> ::= (<instance-definition>)
```

A definstances cannot use classes that have not been previously defined. The instances of a definstances are created in order, and if any individual creation fails, the remainder of the definstances will be aborted. Normally, definstances just use the **make-instance** function (which means delayed Rete activity) to create the instances. However, if this is not desired, then the *active* keyword can be specified after the definstances name so that the **active-make-instance** function will be used.

Example

```
CLIPS> (clear)
CLIPS>
```

```
(defclass A (is-a USER)
    (slot x (default 1)))
CLIPS>
(definstances A-OBJECTS
    (a1 of A)
   (of A (x 65)))
CLIPS> (watch instances)
CLIPS> (reset)
==> instance [a1] of A
==> instance [gen1] of A
CLIPS> (reset)
<== instance [a1] of A
<== instance [gen1] of A
==> instance [a1] of A
==> instance [gen2] of A
CLIPS> (unwatch instances)
CLIPS>
```

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 instance on success or the symbol FALSE on failure. The evaluation of <instance-name-expression> can either be an instance-name, instance-address or a symbol. **initialize-instance** automatically delays all object pattern-matching activities for rules until all slot overrides have been processed. The function **active-initialize-instance** can be used if delayed pattern-matching is not desired.

Syntax

```
(initialize-instance <instance-name-expression>
  <slot-override>*)
```

The **initialize-instance** function performs the following steps in order:

1) All slot-overrides are immediately evaluated and placed via **put-** messages, e.g. (send <instance-name> put-<slot-name> <expression>*).

2) The instance receives the **init** message, e.g. (send <instance-name> init). Normally, the handler attached to class USER will respond to this message. This handler calls the **init-slots** function. This function uses defaults from the class definition (if any) for any slots that do not have slot-overrides. The class defaults are placed directly without the use of messages.

If no slot-override or class default specifies the value of a slot, that value will remain the same. Empty class default values allow **initialize-instance** to clear a slot.

If an error occurs, the instance will *not* be deleted, but the slot values may be in an inconsistent state.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot x (default 34))
   (slot y (default abc))
   (slot z))
CLIPS> (make-instance a of A (y 100))
[a]
CLIPS> (send [a] print)
[a] of A
(x 34)
(y 100)
(z nil)
CLIPS> (send [a] put-x 65)
65
CLIPS> (send [a] put-y abc)
abc
CLIPS> (send [a] put-z "Hello world.")
"Hello world."
CLIPS> (send [a] print)
[a] of A
(x 65)
(y abc)
(z "Hello world.")
CLIPS> (initialize-instance a)
[a]
CLIPS> (send [a] print)
a of A
(x 34)
(y abc)
(z nil)
CLIPS>
```

9.6.3 Reading Slots

Sources external to an object, such as a rule or deffunction, can read an object's slots only by sending the object a message. Message-handlers executing on the behalf of an object can either use messages or direct access to read the object's slots (see section 9.4.2). Several functions also exist which operate implicitly on the active instance for a message that can only be called by message-handlers, such as **dynamic-get**.

Section 12.16 describes ways of testing for the existence of slots and their values.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot x (default abc)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (sym-cat (send [a] get-x) def)
abcdef
CLIPS>
```

9.6.4 Setting Slots

Sources external to an object, such as a rule or deffunction, can write an object's slots only by sending the object a message. Several functions also exist which operate implicitly on the active instance for a message that can only be called by message-handlers, such as **dynamic-put**. The **bind** function can also be used to set a slot's value from within a message-handler.

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
   (slot x (default abc)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] put-x "New value.")
"New value."
CLIPS>
```

9.6.5 Deleting Instances

Sending the **delete** message to an instance removes it from the system. Within a message-handler, the **delete-instance** function (see section 12.16) can be used to delete the active instance for a message.

Syntax

```
(send <instance> delete)
```

9.6.6 Delayed Pattern-Matching When Manipulating Instances

While manipulating instances (either by creating, modifying, or deleting), it is possible to delay pattern-matching activities for rules until after all of the manipulations have been made. This can be accomplished using the **object-pattern-match-delay** function. This function acts identically to the **progn** function, however, any actions that could affect object pattern-matching for rules are delayed until the function is exited. This function's primary purpose is to provide some control over performance.

Syntax

```
(object-pattern-match-delay <action>*)
```

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER))
CLIPS>
(defrule match-A
```

```
  (object (is-a A))
=>)
CLIPS> (make-instance a of A)
[a]
CLIPS> (agenda)
0      match-A: [a]
For a total of 1 activation.
CLIPS> (make-instance b of A)
[b]
CLIPS> (agenda)
0      match-A: [b]
0      match-A: [a]
For a total of 2 activations.
CLIPS>
(object-pattern-match-delay
   (make-instance c of A)
   (println "After c...")
   (agenda)
   (make-instance d of A)
   (println "After d...")
   (agenda))
After c...
0      match-A: [b]
0      match-A: [a]
For a total of 2 activations.
After d...
0      match-A: [b]
0      match-A: [a]
For a total of 2 activations.
CLIPS> (agenda)
0      match-A: [d]
0      match-A: [c]
0      match-A: [b]
0      match-A: [a]
For a total of 4 activations.
CLIPS>
```

9.6.7 Modifying Instances

Four functions are provided for modifying instances. These functions allow instance slot updates to be performed in blocks without requiring a series of put- messages. Each of these functions returns the symbol TRUE if successful, otherwise the symbol FALSE is returned.

9.6.7.1 Directly Modifying an Instance with Delayed Pattern-Matching

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

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot foo)
  (slot bar))
CLIPS> (make-instance a of A)
[a]
CLIPS> (watch all)
CLIPS> (modify-instance a (foo 0))
MSG >> direct-modify ED:1 (<Instance-a> <Pointer-0019CD5A>)
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.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot foo)
  (slot bar))
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>)
TRUE
CLIPS> (unwatch all)
CLIPS>
```

9.6.7.4 Modifying an Instance using Messages with Immediate Pattern-Matching

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

Syntax

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

9.6.8 Duplicating Instances

Four functions are provided for duplicating instances. These functions allow instance duplication and slot updates to be performed in blocks without requiring a series of put- messages. Each of these functions return the instance-name of the new duplicated instance if successful, otherwise the symbol FALSE is returned.

Each of the duplicate functions can optionally specify the name of the instance to which the old instance will be copied. If the name is not specified, the function will generate the name using the (gensym*) function. If the target instance already exists, it will be deleted directly or with a delete message depending on which function was called.

9.6.8.1 Directly Duplicating an Instance with Delayed Pattern-Matching

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

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (setgen 1)
1
CLIPS>
(defclass A (is-a USER)
  (slot foo)
  (slot bar))
CLIPS> (make-instance a of A (foo 0) (bar 4))
[a]
CLIPS> (watch all)
CLIPS> (duplicate-instance a)
MSG >> direct-duplicate ED:1 (<Instance-a> [gen1] <Pointer-00000000>)
HND >> direct-duplicate primary in class USER
       ED:1 (<Instance-a> [gen1] <Pointer-00000000>)
==> instance [gen1] of A
::= local slot foo in instance gen1 <- 0
::= local slot bar in instance gen1 <- 4
HND << direct-duplicate primary in class USER
       ED:1 (<Instance-a> [gen1] <Pointer-00000000>)
MSG << direct-duplicate ED:1 (<Instance-a> [gen1] <Pointer-00000000>)
[gen1]
CLIPS> (unwatch all)
CLIPS>
```

9.6.8.2 Directly Duplicating an Instance with Immediate Pattern-Matching

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

Syntax

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

9.6.8.3 Duplicating an Instance using Messages with Delayed Pattern-Matching

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

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot foo)
  (slot bar))
```

```
CLIPS> (make-instance a of A (foo 0) (bar 4))
[a]
CLIPS> (make-instance b of A)
[b]
CLIPS> (watch all)
CLIPS> (message-duplicate-instance a to b (bar 6))
MSG >> message-duplicate ED:1 (<Instance-a> [b] <Pointer-009F04A0>)
HND >> message-duplicate primary in class USER
       ED:1 (<Instance-a> [b] <Pointer-009F04A0>)
MSG >> delete ED:2 (<Instance-b>)
HND >> delete primary in class USER
       ED:2 (<Instance-b>)
<== 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>
```

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>*)
```

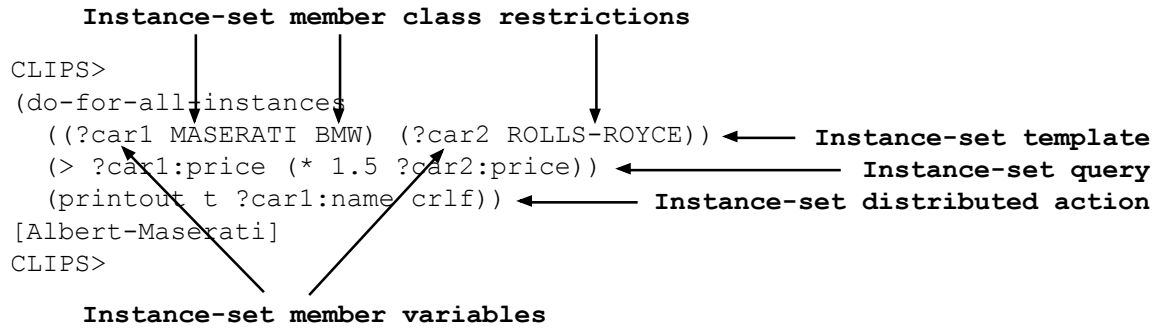
9.7 Instance-set Queries and Distributed Actions

COOL provides a useful query system for determining and performing actions on sets of instances of user-defined classes that satisfy user-defined queries. The instance query system in COOL provides six functions, each of which operate on instance-sets determined by user-defined criteria:

Function	Purpose
any-instancep	Determines if one or more instance-sets satisfy a query
find-instance	Returns the first instance-set that satisfies a query
find-all-instances	Groups and returns all instance-sets which satisfy a query
do-for-instance	Performs an action for the first instance-set which satisfies a query
do-for-all-instances	Performs an action for every instance-set which satisfies a query as they are found
delayed-do-for-all-instances	Groups all instance-sets which satisfy a query and then iterates an action over this group

Explanations on how to form instance-set templates, queries and actions immediately follow, for these definitions are common to all of the query functions. The specific details of each query function will then be given. The following is a complete example of an instance-set query function:

Example



For all of the examples in this section, assume that the commands below have already been entered:

Example

```
CLIPS>
(defclass PERSON (is-a USER)
  (role abstract)
  (slot sex (access read-only)
            (storage shared))
  (slot age (type NUMBER)
            (create-accessor ?NONE)
            (visibility public)))
CLIPS>
(defmessage-handler PERSON put-age (?value)
  (dynamic-put age ?value))
CLIPS>
(defclass FEMALE (is-a PERSON)
  (role abstract)
  (slot sex (source composite)
            (default female)))
CLIPS>
(defclass MALE (is-a PERSON)
  (role abstract)
  (slot sex (source composite)
            (default male)))
CLIPS>
(defclass GIRL (is-a FEMALE)
  (role concrete)
  (slot age (source composite)
            (default 4)
            (range 0.0 17.9)))
CLIPS>
(defclass WOMAN (is-a FEMALE)
  (role concrete)
  (slot age (source composite)
            (default 25)
            (range 18.0 100.0)))
CLIPS>
(defclass BOY (is-a MALE)
  (role concrete)
  (slot age (source composite)
            (default 4)
            (range 0.0 17.9)))
CLIPS>
(defclass MAN (is-a MALE)
  (role concrete)
  (slot age (source composite)
            (default 25)
            (range 18.0 100.0)))
CLIPS>
(definstances PEOPLE
  (Man-1 of MAN (age 18))
  (Man-2 of MAN (age 60))
  (Woman-1 of WOMAN (age 18))
  (Woman-2 of WOMAN (age 60))
  (Woman-3 of WOMAN)
  (Boy-1 of BOY (age 8))
  (Boy-2 of BOY)
  (Boy-3 of BOY)
  (Boy-4 of BOY)
```

```
  (Girl-1 of GIRL (age 8))
  (Girl-2 of GIRL))
CLIPS> (reset)
CLIPS>
```

9.7.1 Instance-set Definition

An **instance-set** is an ordered collection of instances. Each **instance-set member** is an instance of a set of classes, called **class restrictions**, defined by the user. The class restrictions can be different for each instance-set member. The query functions use **instance-set templates** to generate instance-sets. An instance-set template is a set of **instance-set member variables** and their associated class restrictions. Instance-set member variables reference the corresponding members in each instance-set that matches a template. Variables may be used to specify the classes for the instance-set template, but if the constant names of the classes are specified, the classes must already be defined. Module specifiers may be included with the class names; the classes need not be in scope of the current module.

Syntax

```
<instance-set-template>
      ::= (<instance-set-member-template>+)
<instance-set-member-template>
      ::= (<instance-set-member-variable> <class-restrictions>)
<instance-set-member-variable> ::= <single-field-variable>
<class-restrictions>           ::= <class-name-expression>+
```

Example

One instance-set template might be the ordered pairs of boys or men and girls or women.

```
((?man-or-boy BOY MAN) (?woman-or-girl GIRL WOMAN))
```

This instance-set template could have been written equivalently:

```
((?man-or-boy MALE) (?woman-or-girl FEMALE))
```

Instance-set member variables (e.g. ?man-or-boy) are bound to instance-names.

9.7.2 Instance-set Determination

COOL uses straightforward permutations to generate instance-sets that match an instance-set template from the actual instances in the system. The rules are as follows:

1) When there is more than one member in an instance-set template, vary the rightmost members first.

2) When there is more than one class that an instance-set member can be, iterate through the classes from left to right.

3) Examine instances of a class in the order that they were defined.

   a) Recursively examine instances of subclasses in the order that the subclasses were defined. If the specified query class was in scope of the current module, then only subclasses that are also in scope will be examined. Otherwise, only subclasses that are in scope of the module to which the query class belongs will be examined.

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]
```

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

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 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.

```
(println "(" ?man-or-boy "," ?woman-or-girl ")")
```

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.

Example

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

[PRCCODE3] Undefined variable ins referenced in deffunction.

ERROR:
(deffunction MAIN::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. 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 that 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>
```

9.7.8.2 Determining the First Instance-set Satisfying a Query

This function applies a query to each instance-set that 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 that matches the template. Each instance-set that 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>
```

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

This function applies a query to each instance-set that 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))
  (println ?p1 " " ?p2 " " ?p3))
[Girl-2] [Boy-2] [Boy-3]
CLIPS>
```

9.7.8.5 Executing an Action for All Instance-sets Satisfying a Query

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

Syntax

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

Example

Print out all triplets of different people that have the same age. The calls to **str-compare** limit the instance-sets that satisfy the query to combinations instead of permutations. Without these restrictions, two instance-sets that differed only in the order of their members would both satisfy the query.

```
CLIPS>
(do-for-all-instances ((?p1 PERSON) (?p2 PERSON) (?p3 PERSON))
  (and (= ?p1:age ?p2:age ?p3:age)
    (> (str-compare ?p1 ?p2) 0)
    (> (str-compare ?p2 ?p3) 0))
  (println ?p1 " " ?p2 " " ?p3))
```

```
[Girl-2] [Boy-3] [Boy-2]
[Girl-2] [Boy-4] [Boy-2]
[Girl-2] [Boy-4] [Boy-3]
[Boy-4] [Boy-3] [Boy-2]
CLIPS>
```

9.7.8.6 Executing a Delayed Action for All Instance-sets Satisfying a Query

This function is similar to **do-for-all-instances** except that it groups all instance-sets that satisfy the query into an intermediary multifield value. If there are no instance-sets which satisfy the query, then the function returns the symbol FALSE. Otherwise, the specified action is executed for each instance-set in the multifield value, and the return value is the evaluation of the action for the last instance-set to satisfy the query. The intermediary multifield value is discarded. This function can consume large amounts of memory in the same fashion as **find-all-instances**. This function should be used in lieu of **do-for-all-instances** when the action applied to one instance-set would change the result of the query for another instance-set (unless that is the desired effect).

Syntax

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

Example

Delete all boys with the greatest age. The test in this case is another query function that determines if there are any older boys than the one currently being examined. The action needs to be delayed until all boys have been processed, or the greatest age will decrease as the older boys are deleted.

```
CLIPS> (watch instances)
CLIPS>
(delayed-do-for-all-instances ((?b1 BOY))
  (not (any-instancep ((?b2 BOY))
         (> ?b2:age ?b1:age)))
  (send ?b1 delete))
<== instance [Boy-1] of BOY
TRUE
CLIPS> (unwatch instances)
CLIPS> (reset)
CLIPS> (watch instances)
CLIPS>
(do-for-all-instances ((?b1 BOY))
  (not (any-instancep ((?b2 BOY))
         (> ?b2:age ?b1:age)))
  (send ?b1 delete))
<== instance [Boy-1] of BOY
<== instance [Boy-2] of BOY
<== instance [Boy-3] of BOY
<== instance [Boy-4] of BOY
TRUE
```

```
CLIPS> (unwatch instances)
CLIPS>
```

Section 10:
Defmodule Construct

CLIPS provides support for the modular development and execution of knowledge bases with the **defmodule** construct. CLIPS modules allow a set of constructs to be grouped together such that explicit control can be maintained over restricting the access of the constructs by other modules. This type of control is similar to global and local scoping used in languages such as C or Ada (note, however, that the global scoping used by CLIPS is strictly hierarchical and in one direction only—if module A can see constructs from module B, then it is not possible for module B to see any of module A's constructs). By restricting access to deftemplate and defclass constructs, modules can function as blackboards, permitting only certain facts and instances to be seen by other modules. Modules are also used by rules to provide execution control.

10.1 Defining Modules

Modules are defined using the defmodule construct.

Syntax

```
(defmodule <module-name> [<comment>]
   <port-specification>*)

<port-specification> ::= (export <port-item>) |
                         (import <module-name> <port-item>)

<port-item>          ::= ?ALL |
                         ?NONE |
                         <port-construct> ?ALL |
                         <port-construct> ?NONE |
                         <port-construct> <construct-name>+

<port-construct>     ::= deftemplate | defclass |
                         defglobal | deffunction |
                         defgeneric
```

A defmodule cannot be redefined or even deleted once it is defined (with the exception of the MAIN module which can be redefined once). The only way to delete a module is with the **clear** command. Upon startup and after a **clear** command, CLIPS automatically constructs the following defmodule.

```
(defmodule MAIN)
```

All of the predefined system classes (see section 9.2) belong to the MAIN module. However, it is not necessary to import or export the system classes; they are always in scope. Discounting the previous exception, the predefined MAIN module does not import or export any constructs.

However, unlike other modules, the MAIN module can be redefined once after startup or a **clear** command.

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))
```

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>
```

10.3 Specifying Modules

Commands such as **undefrule** and **ppdefrule** require the name of a construct on which to operate. In previous versions of CLIPS, constructs were always referred to by their name only, so it was sufficient just to pass the name of the construct to these commands. With modules, however, it is possible to have a construct with the same name in two different modules. The modules associated with a name can be specified either explicitly or implicitly. To explicitly specify a name's module the module name (a symbol) is listed followed by two colons, ::, and then the name is listed. The module name followed by :: is referred to as a **module specifier**. For example, MAIN::find-stuff, refers to the find-stuff construct in the MAIN module. A module can also be implicitly specified since there is always a "current" module. The current module is changed whenever a defmodule construct is defined or the **set-current-module** function is used. The MAIN module is automatically defined by CLIPS and by default is the current module when CLIPS is started or after a **clear** command is issued. Thus the name find-stuff would implicitly have the MAIN module as its module when CLIPS is first started.

```
CLIPS> (clear)
CLIPS> (defmodule A)
CLIPS> (defglobal A ?*x* = 0)
CLIPS> (defmodule B)
CLIPS> (defglobal B ?*y* = 1)
CLIPS> (ppdefglobal y)
(defglobal B ?*y* = 1)
CLIPS> (ppdefglobal B::y)
(defglobal B ?*y* = 1)
CLIPS> (ppdefglobal x)
[PRNTUTIL1] Unable to find defglobal x.
CLIPS> (ppdefglobal A::x)
(defglobal A ?*x* = 0)
CLIPS>
```

10.4 Importing and Exporting Constructs

Unless specifically **exported** and **imported**, the constructs of one module may not be used by another module. A construct is said to be visible or within scope of a module if that construct can be used by the module. For example, if module *B* wants to use the *foo* deftemplate defined in module *A*, then module *A* must export the *foo* deftemplate and module *B* must import the *foo* deftemplate from module *A*.

```
CLIPS> (clear)
CLIPS> (defmodule A)
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (defmodule B)
CLIPS> (defrule B::bar (foo (x 3)) =>)

[PRNTUTIL2] Syntax Error:  Check appropriate syntax for defrule

ERROR:
(defrule B::bar
   (foo (
```

```
CLIPS> (clear)
CLIPS> (defmodule A (export deftemplate foo))
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (defmodule B (import A deftemplate foo))
CLIPS> (defrule B::bar (foo (x 3)) =>)
CLIPS>
```

CLIPS will not allow a module or other construct to be defined that causes two constructs with the same name to be visible within the same module.

10.4.1 Exporting Constructs

The export specification in a defmodule definition is used to indicate which constructs will be accessible to other modules importing from the module being defined. Only deftemplates, defclasses, defglobals, deffunctions, and defgenerics may be exported. A module may export any valid constructs that are visible to it (not just constructs that it defines).

There are three different types of export specifications. First, a module may export all valid constructs that are visible to it. This accomplished by following the *export* keyword with the *?ALL* keyword. Second, a module may export all valid constructs of a particular type that are visible to it. This accomplished by following the *export* keyword with the name of the construct type followed by the *?ALL* keyword. Third, a module may export specific constructs of a particular type that are visible to it. This accomplished by following the *export* keyword with the name of the construct type followed by the name of one or more visible constructs of the specified type. In the following code, defmodule *A* exports all of its constructs; defmodule *B* exports all of its deftemplates; and defmodule *C* exports the *foo*, *bar*, and *yak* defglobals.

```
(defmodule A (export ?ALL))

(defmodule B (export deftemplate ?ALL))

(defmodule C (export defglobal foo bar yak))
```

The ?NONE keyword may be used in place of the ?ALL keyword to indicate either that no constructs are exported from a module or that no constructs of a particular type are exported from a module.

Defmethods and defmessage-handlers cannot be explicitly exported. Exporting a defgeneric automatically exports all associated defmethods. Exporting a defclass automatically exports all associated defmessage-handlers. Deffacts, definstances, and defrules cannot be exported.

10.4.2 Importing Constructs

The import specification in a defmodule definition is used to indicate which constructs the module being defined will use from other modules. Only deftemplates, defclasses, defglobals, deffunctions, and defgenerics may be imported.

There are three different types of import specifications. First, a module may import all valid constructs that are visible to a specified module. This accomplished by following the *import* keyword with a module name followed by the *?ALL* keyword. Second, a module may import all valid constructs of a particular type that are visible to a specified module. This accomplished by following the *import* keyword with a module name followed by the name of the construct type followed by the *?ALL* keyword. Third, a module may import specific constructs of a particular type that are visible to it. This accomplished by following the *import* keyword with a module name followed by the name of the construct type followed by the name of one or more visible constructs of the specified type. In the following code, defmodule *A* imports all of module *D*'s constructs; defmodule *B* imports all of module *D*'s deftemplates; and defmodule *C* imports the foo, bar, and *yak* defglobals from module *D*.

```
(defmodule A (import D ?ALL))

(defmodule B (import D deftemplate ?ALL))

(defmodule C (import D defglobal foo bar yak))
```

The ?NONE keyword may be used in place of the ?ALL keyword to indicate either that no constructs are imported from a module or that no constructs of a particular type are imported from a module.

Defmethods and defmessage-handlers cannot be explicitly imported. Importing a defgeneric automatically imports all associated defmethods. Importing a defclass automatically imports all associated defmessage-handlers. Deffacts, definstances, and defrules cannot be imported.

A module must be defined before it is used in an import specification. In addition, if specific constructs are listed in the import specification, they must already be defined in the module exporting them. It is not necessary to import a construct from the module in which it is defined in order to use it. A construct can be indirectly imported from a module that directly imports and then exports the module to be used.

10.5 Importing and Exporting Facts and Instances

Facts and instances are "owned" by the module in which their corresponding deftemplate or defclass is defined, *not* by the module which creates them. Facts and instances are thus visible only to those modules that import the corresponding deftemplate or defclass. This allows a knowledge base to be partitioned such that rules and other constructs can only "see" those facts and instances that are of interest to them. Instance names, however, are global in scope, so it is still possible to send messages to an instance of a class that is not in scope.

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>
```

10.5.1 Specifying Instance-Names

Instance-names are required to be unique regardless of the module that owns them. 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 all modules. 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.

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 that is the current focus is executed (note that the **reset** and **clear** commands make the MAIN module the current focus). Rule execution continues until another module becomes the current focus, no rules are left on the agenda, or the **return** function is used from the RHS of a rule. Whenever a module that was focused on runs out of rules on its agenda, the current focus is removed from the focus stack and the next module on the focus stack becomes the current focus. Before a rule executes, the current module is changed to the module in which the executing rule is defined (the current focus). The current focus can be changed by using the **focus** command. See sections 5.2, 5.4.10.2, 12.12, and 13.12 for more details.

Example

```
CLIPS> (clear)
CLIPS> (defmodule MAIN)
CLIPS>
(defrule MAIN::focus-example
  =>
  (println "Firing rule in module MAIN.")
  (focus A B))
CLIPS> (defmodule A)
CLIPS>
(defrule A::example-rule
  =>
  (println "Firing rule in module A."))
CLIPS> (defmodule B)
CLIPS>
(defrule B::example-rule
  =>
  (println "Firing rule in module B."))
CLIPS> (reset)
CLIPS> (run)
Firing rule in module MAIN.
Firing rule in module A.
Firing rule in module B.
CLIPS>
```

Section 11:
Constraint Attributes

This section describes the constraint attributes that can be associated with deftemplates and defclasses so that type checking can be performed on slot values when template facts and instances are created. The constraint information is also analyzed for the patterns on the LHS of a rule to determine if the specified constraints prevent the rule from ever firing.

Two types of constraint checking are supported: static and dynamic. Static constraint checking is always enabled and checks constraint violations when function calls and constructs are parsed. This includes constraint checking between patterns on the LHS of a rule when variables are used in more than one slot. When dynamic constraint checking is enabled, newly created data objects (such as deftemplate facts and instances) have their slot values checked for constraint violations. Essentially, static constraint checking occurs when a CLIPS program is loaded and dynamic constraint checking occurs when a CLIPS program is running. By default, dynamic constraint checking is disabled. It can be enabled using the **set-dynamic-constraint-checking** function.

Unless dynamic constraint checking is enabled, constraint information associated with constructs is not saved when a binary image is created using the **bsave** command.

The general syntax for constraint attributes is shown following.

Syntax

```
<constraint-attribute> ::= <type-attribute> |
                           <allowed-constant-attribute> |
                           <range-attribute> |
                           <cardinality-attribute>
```

11.1 Type Attribute

The type attribute allows the types of values to be stored in a slot to be restricted.

Syntax

```
<type-attribute>     ::= (type <type-specification>)

<type-specification> ::= <allowed-type>+ | ?VARIABLE

<allowed-type>
               ::= SYMBOL | STRING | LEXEME |
                   INTEGER | FLOAT | NUMBER |
                   INSTANCE-NAME | INSTANCE-ADDRESS | INSTANCE |
                   EXTERNAL-ADDRESS | FACT-ADDRESS
```

Using NUMBER for this attribute is equivalent to using both INTEGER and FLOAT. Using LEXEME for this attribute is equivalent to using both SYMBOL and STRING. Using 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 that 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.

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. Type conversion from symbols to strings and strings to symbols is not performed. Similarly, specifying the allowed-numbers attribute is equivalent to specifying constant restrictions on both integers and floats.

11.3 Range Attribute

The range attribute allows a numeric range to be specified for a slot when a numeric value is used in that slot. If a numeric value is not used in that slot, then no checking is performed.

Syntax

```
<range-attribute>     ::= (range <range-specification>
                                 <range-specification>)

<range-specification> ::= <number> | ?VARIABLE
```

Either integers or floats can be used in the range specification with the first value to the range attribute signifying the minimum allowed value and the second value signifying the maximum value. Integers will be temporarily converted to floats when necessary to perform range comparisons. If the keyword ?VARIABLE is used for the minimum value, then the minimum value is negative infinity (-∞). If the keyword ?VARIABLE is used for the maximum value, then the maximum value is positive infinity (+∞). The range attribute cannot be used in conjunction with the allowed-values, allowed-numbers, allowed-integers, or allowed-floats attributes.

11.4 Cardinality Attribute

The cardinality attribute restricts the number of fields that can be stored in a multifield slot. This attribute can not be used with a single field slot.

Syntax

```
<cardinality-attribute>
                  ::= (cardinality <cardinality-specification>
                                   <cardinality-specification>)

<cardinality-specification> ::= <integer> | ?VARIABLE
```

Only integers can be used in the cardinality specification with the first value to the cardinality attribute signifying the minimum number of fields that can be stored in the slot and the second value signifying the maximum number of fields which can be stored in the slot. If the keyword ?VARIABLE is used for the minimum value, then the minimum cardinality is zero. If the keyword ?VARIABLE is used for the maximum value, then the maximum cardinality is positive

infinity (+∞). If the cardinality is not specified for a multifield slot, then it is assumed to be zero to infinity.

The min-number-of-elements and max-number-of-elements attributes found in CLIPS 5.1 are no longer supported. The cardinality attribute should be used in their place.

11.5 Deriving a Default Value From Constraints

Default values for deftemplate and instance slots are automatically derived from the constraints for the slots if an explicit default value is not specified. The following rules are used (in order) to determine the default value for a slot with an unspecified default value.

1) The default type for the slot is chosen from the list of allowed types for the slot in the following order of precedence: SYMBOL, STRING, INTEGER, FLOAT, INSTANCE-NAME, INSTANCE-ADDRESS, FACT-ADDRESS, EXTERNAL-ADDRESS.

2) If the default type has an allowed constant restriction specified (such as the allowed-integers attribute for the INTEGER type), then the first value specified in the allowed constant attribute is chosen as the default value.

3) If the default value was not specified by step 2 and the default type is INTEGER or FLOAT and the range attribute is specified, then the minimum range value is used as the default value if it is not ?VARIABLE, otherwise, the maximum range value is used if it is not ?VARIABLE.

4) If the default value was not specified by step 2 or 3, then the default default value is used. This value is nil for type SYMBOL, "" for type STRING, 0 for type INTEGER, 0.0 for type FLOAT, [nil] for type INSTANCE-NAME, a pointer to a dummy instance for type INSTANCE-ADDRESS, a pointer to a dummy fact for type FACT-ADDRESS, and the NULL pointer for type EXTERNAL-ADDRESS.

5) If the default value is being derived for a single field slot, then the default value derived from steps 1 through 4 is used. The default value for a multifield slot is a multifield value of length zero. However, if the multifield slot has a minimum cardinality greater than zero, then a multifield value with a length of the minimum cardinality is created and the default value that would be used for a single field slot is stored in each field of the multifield value.

11.6 Constraint Violation Examples

The following examples illustrate some of the types of constraint violations that CLIPS can detect.

Example 1

```
CLIPS>
(deftemplate bar
   (slot a (type SYMBOL INTEGER))
   (slot b (type INTEGER FLOAT))
   (slot c (type SYMBOL STRING)))
CLIPS>
(defrule error
  (bar (a ?x))
  (bar (b ?x))
  (bar (c ?x))
  =>)

[RULECSTR1] Variable ?x in CE #3 slot c
has constraint conflicts which make the pattern unmatchable

ERROR:
(defrule MAIN::error-4
   (bar (a ?x))
   (bar (b ?x))
   (bar (c ?x))
   =>)
CLIPS>
```

The first occurrence of the variable ?x in slot a of the first pattern restricts its allowed types to either a symbol or integer. The second occurrence of ?x in slot b of the second pattern further restricts its allowed types to only integers. The final occurence of ?x in the third pattern generates an error because slot c expects ?x to be either a symbol or a string, but its only allowed type is an integer.

Example 2

```
CLIPS>
(deftemplate foo (multislot x (cardinality ?VARIABLE 2)))
CLIPS>
(deftemplate bar (multislot y (cardinality ?VARIABLE 3)))
CLIPS>
(deftemplate woz (multislot z (cardinality 7 ?VARIABLE)))
CLIPS>
(defrule MAIN::error
   (foo (x $?x))
   (bar (y $?y))
   (woz (z $?x $?y))
   =>)

[CSTRNCHK1] The group of restrictions found in CE #3
do not satisfy the cardinality restrictions for slot z

ERROR:
(defrule MAIN::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> (clear)
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 restrictions for
argument #1 of the expression (> ?x 10)
found in CE #2 to be violated

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

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.

Section 12:
Actions And Functions

This section describes various actions and functions which may be used on the LHS and RHS of rules, from the top-level command prompt, and from other constructs such as deffunctions, defmessage-handlers, and defmethods. The terms functions, actions, and commands should be thought of interchangeably. However, when the term **function** is used it generally refers to a function that returns a value. The term **action** refers to a function having no return value but performing some basic operation as a side effect (such as printout). The term **command** refers to functions normally entered at the top-level command prompt (such as the **reset** command, which does not return a value, and the **set-strategy** command, which does return a value).

12.1 Predicate Functions

The following functions perform predicate tests.

12.1.1 Testing For Numbers

The **numberp** function returns the symbol TRUE if its argument is a float or integer, otherwise it returns the symbol FALSE.

Syntax

```
(numberp <expression>)
```

12.1.2 Testing For Floats

The **floatp** function returns the symbol TRUE if its argument is a float, otherwise it returns the symbol FALSE.

Syntax

```
(floatp <expression>)
```

12.1.3 Testing For Integers

The **integerp** function returns the symbol TRUE if its argument is an integer, otherwise it returns the symbol FALSE.

Syntax

```
(integerp <expression>)
```

12.1.4 Testing For Strings Or Symbols

The **lexemep** function returns the symbol TRUE if its argument is a string or symbol, otherwise it returns the symbol FALSE.

Syntax

```
(lexemep <expression>)
```

12.1.5 Testing For Strings

The **stringp** function returns the symbol TRUE if its argument is a string, otherwise it returns the symbol FALSE.

Syntax

```
(stringp <expression>)
```

12.1.6 Testing For Symbols

The **symbolp** function returns the symbol TRUE if its argument is a symbol, otherwise it returns the symbol FALSE. This function may also be called using the name **wordp**.

Syntax

```
(symbolp <expression>)
```

12.1.7 Testing For Even Numbers

The **evenp** function returns the symbol TRUE if its argument is an even number, otherwise it returns the symbol FALSE.

Syntax

```
(evenp <integer-expression>)
```

12.1.8 Testing For Odd Numbers

The **oddp** function returns the symbol TRUE if its argument is an odd number, otherwise it returns the symbol FALSE.

Syntax

```
(oddp <integer-expression>)
```

12.1.9 Testing For Multifield Values

The **multifieldp** function returns the symbol TRUE if its argument is a multifield value, otherwise it returns the symbol FALSE. This function may also be called using the name **sequencep**.

Syntax

```
(multifieldp <expression>)
```

12.1.10 Testing For External-Addresses

The **pointerp** function returns the symbol TRUE if its argument is an external-address, otherwise it returns the symbol FALSE. External-addresses are discussed in further detail in the *Advanced Programming Guide*.

Syntax

```
(pointerp <expression>)
```

12.1.11 Comparing for Equality

The **eq** function returns the symbol TRUE if its first argument is equal in value to all its subsequent arguments, otherwise it returns the symbol FALSE. Note that **eq** compares types as well as values. Thus, (eq 3 3.0) is FALSE since 3 is an integer and 3.0 is a float.

Syntax

```
(eq <expression> <expression>+)
```

Example

```
CLIPS> (eq foo bar mumble foo)
FALSE
CLIPS> (eq foo foo foo foo)
TRUE
CLIPS> (eq 3 4)
FALSE
CLIPS>
```

12.1.12 Comparing for Inequality

The **neq** function returns the symbol TRUE if its first argument is not equal in value to all its subsequent arguments, otherwise it returns the symbol FALSE. Note that **neq** compares types as well as values. Thus, (neq 3 3.0) is TRUE since 3 is an integer and 3.0 is a float.

Syntax

```
(neq <expression> <expression>+)
```

Example

```
CLIPS> (neq foo bar yak bar)
TRUE
CLIPS> (neq foo foo yak bar)
FALSE
CLIPS> (neq 3 a)
TRUE
CLIPS>
```

12.1.13 Comparing Numbers for Equality

The = function returns the symbol TRUE if its first argument is equal in value to all its subsequent arguments, otherwise it returns the symbol FALSE. Note that = compares only numeric values and will convert integers to floats when necessary for comparison.

Syntax

```
(= <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (= 3 3.0)
TRUE
CLIPS> (= 4 4.1)
FALSE
CLIPS>
```

❖ Portability Note

Because the precision of floating point numbers varies from one machine to another, it is possible for the numeric comparison functions to work correctly one machine and incorrectly on another. In fact, you should be aware, even if code is not being ported, that roundoff error can cause erroneous results. For example, the following expression erroneously returns the symbol TRUE because both numbers are rounded up to 0.6666666666666666667.

```
CLIPS> (= 0.6666666666666666666 0.6666666666666666667)
TRUE
CLIPS>
```

12.1.14 Comparing Numbers for Inequality

The <> function returns the symbol TRUE if its first argument is not equal in value to all its subsequent arguments, otherwise it returns the symbol FALSE. Note that <> compares only numeric values and will convert integers to floats when necessary for comparison.

Syntax

```
(<> <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (<> 3 3.0)
FALSE
CLIPS> (<> 4 4.1)
TRUE
CLIPS>
```

❖ Portability Note

See portability note in section 12.1.13.

12.1.15 Greater Than Comparison

The > function returns the symbol TRUE if for all its arguments, argument n-1 is greater than argument n, otherwise it returns the symbol FALSE. Note that > compares only numeric values and will convert integers to floats when necessary for comparison.

Syntax

```
(> <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (> 5 4 3)
TRUE
CLIPS> (> 5 3 4)
FALSE
CLIPS>
```

❖ Portability Note

See portability note in section 12.1.13.

12.1.16 Greater Than or Equal Comparison

The >= function returns the symbol TRUE if for all its arguments, argument n-1 is greater than or equal to argument n, otherwise it returns the symbol FALSE. Note that >= compares only numeric values and will convert integers to floats when necessary for comparison.

Syntax

```
(>= <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (>= 5 5 3)
TRUE
CLIPS> (>= 5 3 5)
FALSE
CLIPS>
```

❖ Portability Note

See portability note in section 12.1.13.

12.1.17 Less Than Comparison

The < function returns the symbol TRUE if for all its arguments, argument n-1 is less than argument n, otherwise it returns the symbol FALSE. Note that < compares only numeric values and will convert integers to floats when necessary for comparison.

Syntax

```
(< <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (< 3 4 5)
TRUE
CLIPS> (< 3 5 4)
FALSE
CLIPS>
```

❖ Portability Note

See portability note in section 12.1.13.

12.1.18 Less Than or Equal Comparison

The <= function returns the symbol TRUE if for all its arguments, argument n-1 is less than or equal to argument n, otherwise it returns the symbol FALSE. Note that <= compares only numeric values and will convert integers to floats when necessary for comparison.

Syntax

```
(<= <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (<= 3 5 5)
TRUE
CLIPS> (<= 5 3 5)
FALSE
CLIPS>
```

❖ Portability Note

See portability note in section 12.1.13.

12.1.19 Boolean And

The **and** function returns the symbol TRUE if each of its arguments evaluates to TRUE, otherwise it returns the symbol FALSE. The **and** function performs short-circuited boolean logic. Each argument of the function is evaluated from left to right. If any argument evaluates to FALSE, then the symbol FALSE is immediately returned by the function.

Syntax

```
(and <expression>+)
```

12.1.20 Boolean Or

The **or** function returns the symbol TRUE if any of its arguments evaluates to TRUE, otherwise it returns the symbol FALSE. The **or** function performs short-circuited boolean logic. Each argument of the function is evaluated from left to right. If any argument evaluates to TRUE, then the symbol TRUE is immediately returned by the function.

Syntax

```
(or <expression>+)
```

12.1.21 Boolean Not

The **not** function returns the symbol TRUE if its argument evaluates to FALSE, otherwise it returns the symbol FALSE.

Syntax

```
(not <expression>)
```

12.2 Multifield Functions

The following functions operate on multifield values.

12.2.1 Creating Multifield Values

This function appends any number of fields together to create a multifield value.

Syntax

```
(create$ <expression>*)
```

The return value of **create$** is a multifield value regardless of the number or types of arguments (single-field or multifield). Calling **create$** with no arguments creates a multifield value of length zero.

Example

```
CLIPS (create$ hammer drill saw screw pliers wrench)
(hammer drill saw screw pliers wrench)
CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))
(7 6 2.0)
CLIPS>
```

12.2.2 Specifying an Element

The **nth$** function will return a specified field from a multifield value.

Syntax

```
(nth$ <integer-expression> <multifield-expression>)
```

where the first argument should be an integer from 1 to the number of elements within the second argument. The symbol **nil** will be returned if the first argument is greater than the number of fields in the second argument.

Example

```
CLIPS> (nth$ 3 (create$ a b c d e f g))
c
CLIPS>
```

12.2.3 Finding an Element

The **member$** function will tell if a single field value is contained in a multifield value.

Syntax

```
(member$ <expression> <multifield-expression>)
```

If the first argument is a single field value and is one of the fields within the second argument, **member$** will return the integer position of the field (from 1 to the length of the second argument). If the first argument is a multifield value and this value is embedded in the second argument, then the return value is a two field multifield value consisting of the starting and ending integer indices of the first argument within the second argument. If neither of these situations is satisfied, then FALSE is returned.

Example

```
CLIPS> (member$ blue (create$ red 3 "text" 8.7 blue))
5
CLIPS> (member$ 4 (create$ red 3 "text" 8.7 blue))
FALSE
CLIPS> (member$ (create$ b c) (create$ a b c d))
(2 3)
CLIPS>
```

12.2.4 Comparing Multifield Values

This function checks if one multifield value is a subset of another; i.e., if all the fields in the first multifield value are also in the second multifield value.

Syntax

```
(subsetp <multifield-expression> <multifield-expression>)
```

If the first argument is a subset of the second argument, the function returns TRUE; otherwise, it returns FALSE. The order of the fields is not considered. If the first argument is bound to a multifield of length zero, the **subsetp** function always returns TRUE.

Example

```
CLIPS> (subsetp (create$ hammer saw drill)
                (create$ hammer drill wrench pliers saw))
```

```
TRUE
CLIPS> (subsetp (create$ wrench crowbar)
                (create$ hammer drill wrench pliers saw))
FALSE
CLIPS>
```

12.2.5 Deletion of Fields in Multifield Values

This function deletes the specified range from a multifield value.

Syntax

```
(delete$ <multifield-expression>
             <begin-integer-expression>
             <end-integer-expression>)
```

The modified multifield value is returned, which is the same as <multifield-expression> with the fields ranging from <begin-integer-expression> to <end-integer-expression> removed. To delete a single field, the begin range field should equal the end range field.

Example

```
CLIPS> (delete$ (create$ hammer drill saw pliers wrench) 3 4)
(hammer drill wrench)
CLIPS> (delete$ (create$ computer printer hard-disk) 1 1)
(printer hard-disk)
CLIPS>
```

12.2.6 Creating Multifield Values from Strings.

This function constructs a multifield value from a string by using each field in a string as a field in a new multifield value.

Syntax

```
(explode$ <string-expression>)
```

A new multifield value is created in which each delimited field in order in <string-expression> is taken to be a field in the new multifield value that is returned. A string with no fields creates a multifield value of length zero. Fields other than symbols, strings, integer, floats, or instances names (such as parentheses or variables) are converted to strings.

Example

```
CLIPS> (explode$ "hammer drill saw screw")
(hammer drill saw screw)
CLIPS> (explode$ "1 2 abc 3 4 \"abc\"  \"def\"")
(1 2 abc 3 4 "abc" "def")
CLIPS> (explode$ "?x ~ )")
```

```
("?x" "~" ")")
CLIPS>
```

12.2.7 Creating Strings from Multifield Values

This function creates a single string from a multifield value.

Syntax

```
(implode$ <multifield-expression>)
```

Each field in <multifield-expression> in order is concatenated into a string value with a single blank separating fields. The new string is returned.

Example

```
CLIPS> (implode$ (create$ hammer drill screwdriver))
"hammer drill screwdriver"
CLIPS> (implode$ (create$ 1 "abc" def "ghi" 2))
"1 "abc" def "ghi" 2"
CLIPS> (implode$ (create$ "abc       def         ghi"))
""abc       def         ghi""
CLIPS>
```

12.2.8 Extracting a Sub-sequence from a Multifield Value

This function extracts a specified range from a multifield value and returns a new multifield value containing just the sub-sequence.

Syntax

```
(subseq$ <multifield-value>
             <begin-integer-expression>
             <end-integer-expression>)
```

where the second and third arguments are integers specifying the begin and end fields of the desired sub-sequence in <multifield-expression>.

Example

```
CLIPS> (subseq$ (create$ hammer drill wrench pliers) 3 4)
(wrench pliers)
CLIPS> (subseq$ (create$ 1 "abc" def "ghi" 2) 1 1)
(1)
CLIPS>
```

12.2.9 Replacing Fields within a Multifield Value

This function replaces a range of field in a multifield value with a series of single-field and/or multifield values and returns a new multifield value containing the replacement values within the original multifield value.

Syntax

```
(replace$ <multifield-expression>
          <begin-integer-expression>
          <end-integer-expression>
          <single-or-multi-field-expression>+)
```

where <begin-integer-expression> to <end-integer-expression> is the range of values to be replaced.

Example

```
CLIPS> (replace$ (create$ drill wrench pliers) 3 3 machete)
(drill wrench machete)
CLIPS> (replace$ (create$ a b c d) 2 3 x y (create$ q r s))
(a x y q r s d)
CLIPS>
```

12.2.10 Inserting Fields within a Multifield Value

This function inserts a series of single-field and/or multifield values at a specified location in a multifield value with and returns a new multifield value containing the inserted values within the original multifield value.

Syntax

```
(insert$ <multifield-expression>
         <integer-expression>
         <single-or-multi-field-expression>+)
```

where <integer-expression> is the location where the values are to be inserted. This value must be greater than or equal to 1. A value of 1 inserts the new value(s) at the beginning of the <multifield-expression>. Any value greater than the length of the <multifield-expression> appends the new values to the end of the <multifield-expression>.

Example

```
CLIPS> (insert$ (create$ a b c d) 1 x)
(x a b c d)
CLIPS> (insert$ (create$ a b c d) 4 y z)
(a b c y z d)
CLIPS> (insert$ (create$ a b c d) 5 (create$ q r))
(a b c d q r)
CLIPS>
```

12.2.11 Getting the First Field from a Multifield Value

This function returns the first field of a multifield value as a multifield value

Syntax

```
(first$ <multifield-expression>)
```

Example

```
CLIPS> (first$ (create$ a b c))
(a)
CLIPS> (first$ (create$))
()
CLIPS>
```

12.2.12 Getting All but the First Field from a Multifield Value

This function returns all but the first field of a multifield value as a multifield value.

Syntax

```
(rest$ <multifield-expression>)
```

Example

```
CLIPS> (rest$ (create$ a b c))
(b c)
CLIPS> (rest$ (create$))
()
CLIPS>
```

12.2.13 Determining the Number of Fields in a Multifield Value

The **length$** function returns an integer indicating the number of fields contained in a multifield value. If the argument passed to **length$** is not the appropriate type, a negative one (-1) is returned.

Syntax

```
(length$ <multifield-expression>)
```

Example

```
CLIPS> (length$ (create$ a b c d e f g))
7
CLIPS>
```

12.2.14 Deleting Specific Values within a Multifield Value

This function deletes specific values contained within a multifield value and returns the modified multifield value.

Syntax

```
(delete-member$ <multifield-expression> <expression>+)
```

where <expression>+ is one or more values to be deleted from <multifield-expression>. If <expression> is a multifield value, the entire sequence must be contained within the first argument in the correct order.

Example

```
CLIPS> (delete-member$ (create$ a b a c) b a)
(c)
CLIPS> (delete-member$ (create$ a b c c b a) (create$ b a))
(a b c c)
CLIPS>
```

12.2.15 Replacing Specific Values within a Multifield Value

This function replaces specific values contained within a multifield value and returns the modified multifield value.

Syntax

```
(replace-member$ <multifield-expression> <substitute-expression>
                 <search-expression>+)
```

where any <search-expression> that is contained within <multifield-expression> is replaced by <substitute-expression>.

Example

```
CLIPS> (replace-member$ (create$ a b a b) (create$ a b a) a b)
(a b a a b a a b a a b a)
CLIPS> (replace-member$ (create$ a b a b) (create$ a b a) (create$ a b))
(a b a a b a)
CLIPS>
```

12.3 String Functions

The following functions perform operations that are related to strings.

12.3.1 String Concatenation

The **str-cat** function will concatenates its arguments into a single string.

Syntax

```
(str-cat <expression>*)
```

Each <expression> should be one of the following types: symbol, string, float, integer, or instance-name.

Example

```
CLIPS> (str-cat "foo" bar)
"foobar"
CLIPS>
```

12.3.2 Symbol Concatenation

The **sym-cat** function will concatenate its arguments into a single symbol. It is functionally identical to the str-cat function with the exception that the returned value is a symbol and not a string.

Syntax

```
(sym-cat <expression>*)
```

Each <expression> should be one of the following types: symbol, string, float, integer, or instance-name.

12.3.3 Taking a String Apart

The **sub-string** function will retrieve a portion of a string from another string.

Syntax

```
(sub-string <integer-expression> <integer-expression>
            <string-expression>)
```

where the first argument, counting from one, must be a number marking the beginning position in the string and the second argument must be a number marking the ending position in the string. If the first argument is greater than the second argument, a null string is returned.

Example

```
CLIPS> (sub-string 3 8 "abcdefghijkl")
```

```
"cdefgh"
CLIPS>
```

12.3.4 Searching a String

The **str-index** function will return the position of a string inside another string.

Syntax

```
(str-index  <lexeme-expression> <lexeme-expression>)
```

where the second argument is searched for the first occurrence of the first argument. The **str-index** function returns the integer starting position, counting from one, of the first argument in the second argument or returns the symbol FALSE if not found.

Example

```
CLIPS> (str-index "def" "abcdefghi")
4
CLIPS> (str-index "qwerty" "qwertypoiuyt")
1
CLIPS> (str-index "qwerty" "poiuytqwer")
FALSE
CLIPS>
```

12.3.5 Evaluating a Function within a String

The **eval** function evaluates the string as though it were entered at the command prompt.

Syntax

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

where the only argument is a string containing the command, constant, or local/global variable to be evaluated. NOTE: **eval** will not evaluate any of the construct definition forms (i.e., **defrule**, **deffacts**, etc.). The return value is the result of the evaluation of the string (or FALSE if an error occurs).

Example

```
CLIPS> (bind ?y 3)
3
CLIPS> (defglobal ?*x* = 4)
CLIPS> (eval "(+ 3 4)")
7
CLIPS> (eval "(+ ?*x* ?y)")
7
CLIPS> (eval "?*x*")
4
CLIPS> (eval "?y")
3
```

```
CLIPS> (eval "3")
3
CLIPS>
```

12.3.6 Evaluating a Construct within a String

The **build** function evaluates the string as though it were entered at the command prompt.

Syntax

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

where the only argument is the construct to be added. The return value is TRUE if the construct was added (or FALSE if 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>
```

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.

Syntax

```
(str-compare <string-or-symbol-expression>
             <string-or-symbol-expression>)
```

This function returns an integer representing the result of the comparison (0 if the strings are equal, < 0 if the first argument < the second argument, and > 0 if the first argument > the second argument).

Example

```
CLIPS> (< (str-compare "string1" "string2") 0)
TRUE   ; since "1" < "2" in ASCII character set
CLIPS> (str-compare "abcd" "abcd")
0
CLIPS>
```

12.3.10 Determining the Length of a String

The **str-length** function returns the length of a string as an integer.

Syntax

```
(str-length <string-or-symbol-expression>)
```

Example

```
CLIPS> (str-length "abcd")
4
CLIPS> (str-length xyz)
```

```
3
CLIPS>
```

12.3.11 Checking the Syntax of a Construct or Function Call within a String

The function **check-syntax** allows the text representation of a construct or function call to be checked for syntax and semantic errors.

Syntax

```
(check-syntax <construct-or-function-string>)
```

This function returns FALSE if there are no errors or warnings encountered parsing the construct or function call. The symbol MISSING-LEFT-PARENTHESIS is returned if the first token is not a left parenthesis. The symbol EXTRANEOUS-INPUT-AFTER-LAST-PARENTHESIS is returned if there are additional tokens after the closing right parenthesis of the construct or function call. If errors or warnings are encounted parsing, the a multifield of length two is returned. The first field of the multifield is a string containing the text of the error message (or the symbol FALSE if no errors were encountered). The second field of the multifield is a string containing the text of the warning message (or the symbol FALSE if no warnings were encountered).

Example

```
CLIPS> (check-syntax "(defrule example =>)")
FALSE
CLIPS> (check-syntax "(defrule foo (number 40000000000000000000) =>)")
(FALSE "[SCANNER1] WARNING: Over or underflow of long long integer.
")
CLIPS> (check-syntax "(defrule example (3) =>)")
("
[PRNTUTIL2] Syntax Error:  Check appropriate syntax for the first field of a
pattern.

ERROR:
(defrule MAIN::example
   (3
" FALSE)
CLIPS>
```

12.3.12 Converting a String to a Field

The **string-to-field** function parses a string and converts its contents to a primitive data type.

Syntax

```
(string-to-field  <string-or-symbol-expression>)
```

where the only argument is the string to be parsed. Essentially calling **string-to-field** with its string argument is equivalent to calling the **read** function and manually typing the contents of the string argument or reading it from a file.

Example

```
CLIPS> (string-to-field "3.4")
3.4
CLIPS> (string-to-field "a b")
a
CLIPS>
```

12.4 The CLIPS I/O System

CLIPS uses a system called I/O routers to provide very flexible I/O while remaining portable. A more complete discussion of I/O routers is covered in the *Advanced Programming Guide*.

12.4.1 Logical Names

One of the key concepts of I/O routing is the use of logical names. Logical names allow reference to an I/O device without having to understand the details of the implementation of the reference. Many functions in CLIPS make use of logical names. A logical name can be either a 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 out. The printout and format functions write to stdout if t is specified as the logical name.
werror	All error messages are sent to this logical name.
wwarning	All warning messages are sent to this logical name.

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

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	Character read access. Specified file must exist.
w	Character write access. Existing content overwritten.
a	Character write access. Writes append to end of file.
r+	Read and write access. Specified file must exist.
w+	Read and write access. Existing content overwritten.
a+	Read and write access. Writes append to end of file.

Binary character mode can also be specified by appending a 'b' at the end of the mode or immediately preceding the '+' character (e.g. rb, rb+, or r+b). If the mode is not specified, it defaults to character read access.

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 "output.txt" writeFile "w")
TRUE
CLIPS> (open "input.txt" 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.

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 "output.txt" writeFile "w")
TRUE
CLIPS> (open "input.txt" readFile)
TRUE
CLIPS> (close writeFile)
TRUE
CLIPS> (close writeFile)
FALSE
CLIPS> (close)
TRUE
CLIPS> (close)
FALSE
CLIPS>
```

12.4.2.3 Printout, Print, and Println

The function **printout** allows output to a device attached to a logical name. The logical name *must* be specified and the device must have been prepared previously for output (e.g., a file must be opened first). To send output to **stdout**, use a **t** for the logical name. If the logical name **nil** is used, the **printout** function does nothing.

Syntax

```
(printout <logical-name> <expression>*)
(print <expression>*)
(println <expression>*)
```

Any number of expressions may be placed in a **printout** to be printed. Each expression is evaluated and printed (with no spaces added between each printed expression). The symbol **crlf** used as an <expression> will force a carriage return/newline and may be placed anywhere in the list of expressions to be printed. Similarly, the symbols **tab**, **vtab**, and **ff** will print respectively a tab, a vertical tab, and a form feed. The appearance of these special symbols may vary from one operating system to another. The printout function strips quotation marks from around strings when it prints them. Fact-addresses, instance-addresses and external-addresses can be printed by the printout function. This function has no return value.

The **print** and **println** functions are variants of the **printout** function. Both function always direct output to **stdout** and the **println** function appends a carriage return/line feed after printing all of its arguments.

Example

```
CLIPS> (printout t "Hello there!" crlf)
Hello There!
CLIPS> (println "Hello There!")
Hello There!
CLIPS> (print "Hello There!" crlf)
Hello There!
CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS>
```

12.4.2.4 Read

The **read** function allows a user to input information for a single field. All of the standard field rules (e.g., multiple symbols must be embedded within quotes) apply.

Syntax

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

where <logical-name> is an optional parameter. If specified, **read** tries to read from whatever is attached to the logical file name. If <logical-name> is **t** or is not specified, the function will read from **stdin**. All the delimiters defined in section 2.3.1 can be used as delimiters. The **read** function always returns a primitive data type. Spaces, carriage returns, and tabs only act as delimiters and are not contained within the return value (unless these characters are included within double quotes as part of a string). If an end of file (EOF) is encountered while reading, **read** will return the symbol **EOF**. If errors are encountered while reading, the string "*** READ ERROR ***" will be returned.

Example

```
CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" mydata)
TRUE
CLIPS> (read mydata)
red
CLIPS> (read mydata)
green
CLIPS> (read mydata)
```

```
EOF
CLIPS> (close)
TRUE
CLIPS>
```

12.4.2.5 Readline

The **readline** function is similar to the **read** function, but it allows a whole string to be input instead of a single field. Normally, **read** will stop when it encounters a delimiter. The **readline** function only stops when it encounters a carriage return, a semicolon, or an EOF. Any tabs or spaces in the input are returned by **readline** as a part of the string. The **readline** function returns a string.

Syntax

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

where <logical-name> is an optional parameter. If specified, **readline** tries to read from whatever is attached to the logical file name. If <logical-name> is **t** or is not specified, the function will read from **stdin**. As with the **read** function, if an EOF is encountered, **readline** will return the symbol EOF. If an error is encountered during input, **readline** returns the string "*** READ ERROR ***".

Example

```
CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" mydata)
TRUE
CLIPS> (readline mydata)
"red green"
CLIPS> (readline mydata)
EOF
CLIPS> (close)
TRUE
CLIPS>
```

12.4.2.6 Format

The **format** function allows a user to send formatted output to a device attached to a logical name. It can be used in place of **printout** when special formatting of output information is desired. Although a slightly more complicated function, **format** provides much better control over how the output is formatted. The format commands are similar to the **printf** statement in C. The **format** function always returns a string containing the formatted output. A logical name of **nil** may be used when the formatted return string is desired without writing to a device.

Syntax

```
(format <logical-name> <string-expression> <expression>*)
```

If **t** is given, output is sent to **stdout**. The second argument to format, called the control string, specifies how the output should be formatted. Subsequent arguments to format (the parameter list for the control string) are the expressions which are to be output as indicated by the control string. **Format** currently does not allow expressions returning multifield values to be included in the parameter list.

The control string consists of text and format flags. Text is output exactly as specified, and format flags describe how each parameter in the parameter list is to be formatted. The first format flag corresponds to the first value in the parameter list, the second flag corresponds to the second value, etc. The format flags must be preceded by a percent sign (%) and are of the general format

%-M.Nx

where x is one of the flags listed below, the minus sign is an optional justification flag, and M and N are optional parameters which specify the field width and the precision argument (which varies in meaning based on the format flag). If M is used, at least M characters will be output. If more than M characters are required to display the value, **format** expands the field as needed. If M starts with a 0 (e.g., %07d), a zero is used as the pad character; otherwise, spaces are used. If N is not specified, it defaults to six digits for floating-point numbers. If a minus sign is included before the M, the value will be left justified; otherwise the value is right justified.

Format Flag	Meaning
c	Display parameter as a single character.
d	Display parameter as a long long integer. (The N specifier is the minimum number of digits to be printed.)
f	Display parameter as a floating-point number (The N specifier is the number of digits following the decimal point).
e	Display parameter as a floating-point using power of 10 notation (The N specifier is the number of digits following the decimal point).
g	Display parameter in the most general format, whichever is shorter (the N specifier is the number of significant digits to be printed).
o	Display parameter as an unsigned octal number. (The N specifier is the minimum number of digits to be printed.)
x	Display parameter as an unsigned hexadecimal number. (The N specifier is the minimum number of digits to be printed.)
s	Display parameter as a string. Strings will have the leading and trailing quotes stripped. (The N specifier indicates the maximum number of characters to be printed. Zero also cannot be used for the pad character.)
n	Put a new line in the output.
r	Put a carriage return in the output.
%	Put the percent character in the output.

Example

```
CLIPS> (format t "Hello World!%n")
Hello World!
"Hello World!
"
CLIPS> (format nil "Integer:     |%d|" 12)
"Integer:     |12|"
CLIPS> (format nil "Integer:     |%4d|" 12)
"Integer:     |  12|"
CLIPS> (format nil "Integer:     |%-04d|" 12)
"Integer:     |12  |"
CLIPS> (format nil "Integer:     |%6.4d|" 12)
"Integer:     |  0012|"
CLIPS> (format nil "Float:       |%f|" 12.01)
"Float:       |12.010000|"
CLIPS> (format nil "Float:       |%7.2f| "12.01)
"Float:       |  12.01| "
CLIPS> (format nil "Test:        |%e|" 12.01)
"Test:        |1.201000e+01|"
CLIPS> (format nil "Test:        |%7.2e|" 12.01)
"Test:        |1.20e+01|"
CLIPS> (format nil "General:     |%g|" 1234567890)
```

```
"General:     |1.23457e+09|"
CLIPS> (format nil "General:     |%6.3g|" 1234567890)
"General:     |1.23e+09|"
CLIPS> (format nil "Hexadecimal: |%x|" 12)
"Hexadecimal: |c|"
CLIPS> (format nil "Octal:       |%o|" 12)
"Octal:       |14|"
CLIPS> (format nil "Symbols:     |%s| |%s|" value-a1 capacity)
"Symbols:     |value-a1| |capacity|"
CLIPS>
```

❖ Portability Note

The **format** function uses the C function **sprintf** as a base. Some systems may not support **sprintf** or may not support all of these features, which may affect how **format** works.

12.4.2.7 Rename

The **rename** function is used to change the name of a file.

Syntax

```
(rename <old-file-name> <new-file-name>)
```

Both <old-file-name> and <new-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 either <old-file-name> or <new-file-name> must be escaped with a backslash. The **rename** function returns TRUE if it was successful, otherwise FALSE.

❖ Portability Note

The **rename** function uses the ANSI C function **rename** as a base.

12.4.2.8 Remove

The **remove** function is used to delete a file.

Syntax

```
(remove <file-name>)
```

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 **remove** function returns TRUE if it was successful, otherwise FALSE.

❖ Portability Note

The **remove** function uses the ANSI C function **remove** as a base.

12.4.2.9 Get Character

The **get-char** function allows a single character to be retrieved from a logical name.

Syntax

```
(get-char [<logical-name>])
```

where <logical-name> is an optional parameter. If specified, **get-char** tries to retrieve a character from the specified logical file name. If <logical-name> is **t** or is not specified, the function will read from **stdin**. The return value is the integer ASCII value of the character retrieved. The value of -1 is returned if the end of file is encountered while retrieving a character.

Example

```
CLIPS> (open example.txt example "w")
TRUE
CLIPS> (printout example "ABC" crlf)
CLIPS> (close example)
TRUE
CLIPS> (open example.txt example)
TRUE
CLIPS> (get-char example)
65
CLIPS> (format nil "%c" (get-char example))
"B"
CLIPS> (get-char example)
67
CLIPS> (get-char example)
10
CLIPS> (get-char example)
-1
CLIPS>
(progn (print "Press any character to continue...")
       (get-char t))
Press any character to continue...
13
CLIPS> (close)
TRUE
CLIPS>
```

12.4.2.10 Unget Character

The **unget-char** function allows a single character to be put back to a logical name.

Syntax

```
(unget-char [<logical-name>] <character>)
```

where <logical-name> is an optional parameter and <character> is an integer ASCII value of a character. If <logical-name> is specified, **unget-char** tries to put back <character> to the specified logical file name. If <logical-name> is **t** or is not specified, the function will put back the character to **stdin**. If successful, the return value is the integer ASCII value of the character put back; otherwise, -1 is returnd.

Example

```
CLIPS> (open example.txt example "w")
TRUE
CLIPS> (printout example "ABC" crlf)
CLIPS> (close example)
TRUE
CLIPS> (open example.txt example)
TRUE
CLIPS> (get-char example)
65
CLIPS> (unget-char example 65)
65
CLIPS> (get-char example)
65
CLIPS> (get-char example)
66
CLIPS> (unget-char example 68)
68
CLIPS> (get-char example)
68
CLIPS> (get-char example)
67
CLIPS> (close)
TRUE
CLIPS>
```

12.4.2.11 Read Number

The **read-number** function allows a user to input a single number using the localized format (if one has been specified using the set-locale function). If a localized format has not been specified, then the C format for a number is used.

Syntax

```
(read-number [<logical-name>])
```

where <logical-name> is an optional parameter. If specified, **read-number** tries to read from whatever is attached to the logical file name. If <logical-name> is **t** or is not specified, the function will read from **stdin**. If a number is successfully parsed, the **read-number** function will return either an integer (if the number contained just a sign and digits) or a float (if the number contained the localized decimal point character or an exponent). If an end of file (EOF) is encountered while reading, **read-number** will return the symbol **EOF**. If errors are encountered while reading, the string "*** READ ERROR ***" will be returned.

Example

```
CLIPS> (read-number)
34
34
CLIPS> (read-number)
34.0
34.0
CLIPS> (read-number)
23,0
"*** READ ERROR ***"
CLIPS>
```

12.4.2.12 Set Locale

The **set-locale** function allows a user to specify a locale which affects the numeric format behavior of the **format** and **read-number** functions. Before a number is printed by the **format** function or is parsed by the **read-number** function, the locale is temporarily changed to the last value specified to the **set-locale** function (or the default C locale if no value was previously specified).

Syntax

```
(set-locale [<locale-string>])
```

where the optional argument <locale-string> is a string containing the new locale to be used by the **format** and **read-number** functions. If <local-string> is specified, then the value of the previous locale is returned. If <locale-string> is not specified, then the value of the current locale is returned. A <locale-string> value of "" uses the native locale (and the specification of this locale is dependent on the environment in which CLIPS is run). A <locale-string> of "C" specifies the standard C locale (which is the default).

Example

```
;;; This example assumes that the native
;;; locale has been set to Germany.
CLIPS> (read-number)
3.21
3.21
CLIPS> (read-number)
3,21
"*** READ ERROR ***"
CLIPS> (format nil "%f" 3.1)
"3.100000"
CLIPS> (set-locale "")
"C"
CLIPS> (read-number)
3.21
"*** READ ERROR ***"
CLIPS> (read-number)
3,21
3.21
CLIPS> (format nil "%f" 3.1)
```

```
"3,100000"
CLIPS>
```

❖ Portability Note

The CLIPS **set-locale** function uses the ANSI C function **setlocale** to temporarily change the locale. Setting the native locale used by the setlocale function when <local-string> is specified as the empty string "" varies from one operating system to another. For example, in Windows 7 the native locale is set by clicking on the Start menu, selecting Control Panel, selecting Region and Language, selecting the Formats tab, and then selecting a region from the drop-down menu such as German (Germany). Alternately in Windows, the <local-string> could be specified as "DE" for Germany.

In Mac OS X, the native local can be specified by launching System Preferences, clicking on Language & Region, and then setting the region to Germany. However, this only works when running CLIPS from the Terminal application. Alternately the <locale-string> could be specified as "de_DE" to specify Germany for either the GUI or Terminal version of CLIPS.

12.4.2.13 Flush

The **flush** function writes pending output to a file stream previously opened with the **open** command. The file is specified by a logical name previously attached to the desired stream. Flushed files remain open. Typically output does not need to be flushed since closing the file with the **close** command will flush pending output.

Syntax

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

If **flush** is called without arguments, all open files will be flushed. The **flush** function returns TRUE if any files were successfully closed, otherwise FALSE is returned.

❖ Portability Note

The **flush** function uses the ANSI C function **fflush** as a base.

12.4.2.14 Rewind

The **rewind** function sets the position in a file stream previously opened with the **open** command to the beginning of the file. The file is specified by a logical name previously attached to the desired stream.

Syntax

```
(rewind <logical-name>)
```

The **rewind** function returns TRUE if the specified logical name exists, otherwise FALSE is returned.

❖ Portability Note

The **rewind** function uses the ANSI C function **rewind** as a base.

12.4.2.15 Tell

The **tell** function returns the current position in a file stream previously opened with the **open** command. The file is specified by a logical name previously attached to the desired stream.

Syntax

```
(tell <logical-name>)
```

The **tell** function returns the integer file position if successful, otherwise FALSE is returned if the logical name does not exist or an error occurs.

❖ Portability Note

The **tell** function uses the ANSI C function **ftell** as a base.

12.4.2.16 Seek

The **seek** function sets the current position in a file stream previously opened with the **open** command. The file is specified by a logical name previously attached to the desired stream.

Syntax

```
(seek <logical-name> <offset> <relative-position)
```

The <offset> argument is an integer value and the <relative-position> argument is one of the symbols **seek-set**, **seek-cur**, or **seek-end**. If **seek-set** is specified, the file position is set based on the offset relative to the beginning of the file. If **seek-cur** is specified, the file position is set based on the offset relative to the current positiion in the file. If **seek-end** is specified, the file position is set based on the offset relative to the end of the file. The **seek** function returns TRUE if the file position was successfully set, otherwise FALSE is returned if the logical name does not exist or an error occurs.

❖ Portability Note

The **seek** function uses the ANSI C function **fseek** as a base.

12.5 Math Functions

CLIPS provides several functions for mathematical computations. They are split into two packages: a set of standard math functions and a set of extended math functions.

12.5.1 Standard Math Functions

The standard math functions are listed below. These functions should be used only on numeric arguments. An error message will be printed if a string argument is passed to a math function.

12.5.1.1 Addition

The + function returns the sum of its arguments. Each of its arguments should be a numeric expression. Addition is performed using the type of the arguments provided unless mixed mode arguments (integer and float) are used. In this case, the function return value and integer arguments are converted to floats after the first float argument has been encountered. This function returns a float if any of its arguments is a float, otherwise it returns an integer.

Syntax

```
(+ <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (+ 2 3 4)
9
CLIPS> (+ 2 3.0 5)
10.0
CLIPS> (+ 3.1 4.7)
7.8
CLIPS>
```

12.5.1.2 Subtraction

The - function returns the value of the first argument minus the sum of all subsequent arguments. Each of its arguments should be a numeric expression. Subtraction is performed using the type of the arguments provided unless mixed mode arguments (integer and float) are used. In this case, the function return value and integer arguments are converted to floats after the first float argument has been encountered. This function returns a float if any of its arguments is a float, otherwise it returns an integer.

Syntax

```
(- <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (- 12 3 4)
5
CLIPS> (- 12 3.0 5)
4.0
CLIPS> (- 4.7 3.1)
1.6
CLIPS>
```

12.5.1.3 Multiplication

The * function returns the product of its arguments. Each of its arguments should be a numeric expression. Multiplication is performed using the type of the arguments provided unless mixed mode arguments (integer and float) are used. In this case, the function return value and integer arguments are converted to floats after the first float argument has been encountered. This function returns a float if any of its arguments is a float, otherwise it returns an integer.

Syntax

```
(* <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (* 2 3 4)
24
CLIPS> (* 2 3.0 5)
30.0
CLIPS> (* 3.1 4.7)
14.57
CLIPS>
```

12.5.1.4 Division

The / function returns the value of the first argument divided by each of the subsequent arguments. Each of its arguments should be a numeric expression. Each argument is *automatically converted* to a float and floating point division is performed. This function returns a float.

Syntax

```
(/ <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (/ 4 2)
```

```
2.0
CLIPS> (/ 4.0 2.0)
2.0
CLIPS> (/ 24 3 4)
2.0
CLIPS>
```

12.5.1.5 Integer Division

The **div** function returns the value of the first argument divided by each of the subsequent arguments. Each of its arguments should be a numeric expression. Each argument is *automatically converted* to an integer and integer division is performed. This function returns an integer.

Syntax

```
(div <numeric-expression> <numeric-expression>+)
```

Example

```
CLIPS> (div 4 2)
2
CLIPS> (div 5 2)
2
CLIPS> (div 33 2 3 5)
1
CLIPS>
```

12.5.1.6 Maximum Numeric Value

The **max** function returns the value of its largest numeric argument. Each of its arguments should be a numeric expression. When necessary, integers are temporarily converted to floats for comparison. The return value will either be integer or float (depending upon the type of the largest argument).

Syntax

```
(max <numeric-expression>+)
```

Example

```
CLIPS> (max 3.0 4 2.0)
4
CLIPS>
```

12.5.1.7 Minimum Numeric Value

The **min** function returns the value of its smallest numeric argument. Each of its arguments should be a numeric expression. When necessary, integers are temporarily converted to floats for comparison. The return value will either be integer or float (depending upon the type of the smallest argument).

Syntax

```
(min <numeric-expression>+)
```

Example

```
CLIPS> (min 4 0.1 -2.3)
-2.3
CLIPS>
```

12.5.1.8 Absolute Value

The **abs** function returns the absolute value of its only argument (which should be a numeric expression). The return value will either be integer or float (depending upon the type the argument).

Syntax

```
(abs <numeric-expression>)
```

Example

```
CLIPS> (abs 4.0)
4.0
CLIPS> (abs -2)
2
CLIPS>
```

12.5.1.9 Convert To Float

The **float** function converts its only argument (which should be a numeric expression) to type float and returns this value.

Syntax

```
(float <numeric-expression>)
```

Example

```
CLIPS> (float 4.0)
4.0
CLIPS> (float -2)
```

```
-2.0
CLIPS>
```

12.5.1.10 Convert To Integer

The **integer** function converts its only argument (which should be a numeric expression) to type integer and returns this value.

Syntax

```
(integer <numeric-expression>)
```

Example

```
CLIPS> (integer 4.0)
4
CLIPS> (integer -2)
-2
CLIPS>
```

12.5.2 Extended Math Functions

In addition to standard math functions, CLIPS also provides a large number of scientific and trigonometric math functions for more extensive computations. Although included in the generic version of CLIPS, if an expert system does not need these capabilities, these functions may be excluded from the executable element of CLIPS to provide more memory (see the *Advanced Programming Guide*).

12.5.2.1 Trigonometric Functions

The following trigonometric functions take one numeric argument and return a floating-point number. The argument is expected to be in radians.

Function	Returns
acos	arccosine
acot	arccotangent
acsc	arccosecant
asec	arcsecant
asin	arcsine
atan	arctangent
cos	cosine

Function	Returns
acosh	hyperbolic arccosine
acoth	hyperbolic arccotangent
acsch	hyperbolic arccosecant
asech	hyperbolic arcsecant
asinh	hyperbolic arcsine
atanh	hyperbolic arctangent
cosh	hyperbolic cosine

cot	cotangent
csc	cosecant
sec	secant
sin	sine
tan	tangent

coth	hyperbolic cotangent
csch	hyperbolic cosecant
sech	hyperbolic secant
sinh	hyperbolic sine
tanh	hyperbolic tangent

Example

```
CLIPS> (cos 0)
1.0
CLIPS> (acos 1.0)
0.0
CLIPS>
```

12.5.2.2 Convert From Degrees to Grads

The **deg-grad** function converts its only argument (which should be a numeric expression) from units of degrees to units of grads (360 degrees = 400 grads). The return value of this function is a float.

Syntax

```
(deg-grad <numeric-expression>)
```

Example

```
CLIPS> (deg-grad 90)
100.0
CLIPS>
```

12.5.2.3 Convert From Degrees to Radians

The **deg-rad** function converts its only argument (which should be a numeric expression) from units of degrees to units of radians (360 degrees = 2π radians). The return value of this function is a float.

Syntax

```
(deg-rad <numeric-expression>)
```

Example

```
CLIPS> (deg-rad 180)
3.141592653589793
CLIPS>
```

12.5.2.4 Convert From Grads to Degrees

The **grad-deg** function converts its only argument (which should be a numeric expression) from units of grads to units of degrees (360 degrees = 400 grads). The return value of this function is a float.

Syntax

```
(grad-deg <numeric-expression>)
```

Example

```
CLIPS> (grad-deg 100)
90.0
CLIPS>
```

12.5.2.5 Convert From Radians to Degrees

The **rad-deg** function converts its only argument (which should be a numeric expression) from units of radians to units of degrees (360 degrees = 2π radians). The return value of this function is a float.

Syntax

```
(rad-deg <numeric-expression>)
```

Example

```
CLIPS> (rad-deg 3.141592653589793)
180.0
CLIPS>
```

12.5.2.6 Return the Value of π

The **pi** function returns the value of π (3.141592653589793...) as a float.

Syntax

```
(pi)
```

Example

```
CLIPS> (pi)
3.141592653589793
CLIPS>
```

12.5.2.7 Square Root

The **sqrt** function returns the square root of its only argument (which should be a numeric expression) as a float.

Syntax

```
(sqrt <numeric-expression>)
```

Example

```
CLIPS> (sqrt 9)
3.0
CLIPS>
```

12.5.2.8 Power

The ****** function raises its first argument to the power of its second argument and returns this value as a float.

Syntax

```
(** <numeric-expression> <numeric-expression>)
```

Example

```
CLIPS> (** 3 2)
9.0
CLIPS>
```

12.5.2.9 Exponential

The **exp** function raises the value e (the base of the natural system of logarithms, having a value of approximately 2.718...) to the power specified by its only argument and returns this value as a float.

Syntax

```
(exp <numeric-expression>)
```

Example

```
CLIPS> (exp 1)
2.718281828459045
CLIPS>
```

12.5.2.10 Logarithm

Given n (the only argument) and the value e is the base of the natural system of logarithms, the **log** function returns the float value x such that the following equation is satisfied:

$$n = e^{x}$$

Syntax

```
(log <numeric-expression>)
```

Example

```
CLIPS> (log 2.718281828459045)
1.0
CLIPS>
```

12.5.2.11 Logarithm Base 10

Given n (the only argument), the **log10** function returns the float value x such that the following equation is satisfied:

$$n = 10^{x}$$

Syntax

```
(log10 <numeric-expression>)
```

Example

```
CLIPS> (log10 100)
2.0
CLIPS>
```

12.5.2.12 Round

The **round** function rounds its only argument (which should be a numeric expression) toward the closest integer. If the argument is exactly between two integers, it is rounded down. The return value of this function is an integer.

Syntax

```
(round <numeric-expression>)
```

Example

```
CLIPS> (round 3.6)
```

```
4
CLIPS>
```

12.5.2.13 Modulus

The **mod** function returns the remainder of the result of dividing its first argument by its second argument (assuming that the result of division must be an integer). It returns an integer if both arguments are integers, otherwise it returns a float.

Syntax

```
(mod <numeric-expression> <numeric-expression>)
```

Example

```
CLIPS> (mod 5 2)
1
CLIPS> (mod 3.7 1.2)
0.1
CLIPS>
```

12.6 Procedural Functions

The following are functions which provide procedural programming capabilities as found in languages such as Pascal, C and Ada.

12.6.1 Binding Variables

Occasionally, it is important to create new variables or to modify the value of previously bound variables on the RHS of a rule. The **bind** function provides this capability.

Syntax

```
(bind <variable> <expression>*)
```

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> (bind ?x 32)
32
CLIPS> ?x
32
CLIPS> (reset)
CLIPS> ?x
[EVALUATN1] Variable x is unbound
FALSE
CLIPS>
```

Example 2

```
CLIPS>
(defclass A (is-a USER)
   (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>
```

12.6.2 If...then...else Function

The **if** function provides an **if...then...else** structure to allow for conditional execution of a set of actions.

Syntax

```
(if <expression>
   then
      <action>*
   [else
      <action>*])
```

Any number of allowable actions may be used inside of the **then** or **else** portion, including another **if...then...else** structure. The **else** portion is optional. If <expression> evaluates to anything other than the symbol FALSE, then the actions associated with the **then** portion are executed. Otherwise, the actions associated with the **else** portion are executed. The return value of the if function is the value of the last <expression> or <action> evaluated.

Example

```
(defrule closed-valves
   (temp (value high))
   (valve (id ?v) (state closed))
   =>
   (if (= ?v 6)
      then
      (println "The special valve " ?v " is closed!")
      (assert (perform special operation))
      else
      (println "Valve " ?v " is normally closed")))
```

Note that this rule could have been accomplished just as easily with two rules, and that it is usually better to accomplish this with two rules.

```
(defrule closed-valves-number-6
   (temp (value high))
   (valve (id 6) (state closed))
   =>
   (println "The special valve 6 is closed!")
   (assert (perform special operation)))

(defrule closed-valves-other-than-6
   (temp (value high))
   (valve (id ?v&~6) (state closed))
   =>
   (println "Valve " ?v " is normally closed"))
```

12.6.3 While

The **while** function is provided to allow simple looping. Its use is similar to that of the **if** function.

Syntax

```
(while <expression> [do]
   <action>*)
```

Again, all predicate functions are available for use in **while**. Any number of allowable actions may be placed inside the **while** block, including **if...then...else** or additional **while** structures. The test is performed prior to the first execution of the loop. The actions within the **while** loop are executed until <expression> evaluates to the symbol FALSE. The **while** may optionally include the symbol **do** after the condition and before the first action. The **break** and **return** functions can be used to terminate the loop prematurely. The return value of this function is FALSE unless the **return** function is used to terminate the loop.

Example

```
(defrule open-valves
   (valves-open-through (id ?v))
   =>
   (while (> ?v 0)
      (println "Valve " ?v " is open")
      (bind ?v (- ?v 1))))
```

12.6.4 Loop-for-count

The **loop-for-count** function is provided to allow simple iterative looping.

Syntax

```
(loop-for-count <range-spec> [do] <action>*)

<range-spec>  ::= <end-index> |
                        (<loop-variable> <start-index> <end-index>) |
                        (<loop-variable> <end-index>)
<start-index> ::= <integer-expression>
<end-index>   ::= <integer-expression>
```

Performs the given actions the number of times specified by <range-spec>. If <start-index> is not given, it is assumed to be one. If <start-index> is greater than <end-index>, then the body of the loop is never executed. The integer value of the current iteration can be examined with the loop variable, if specified.The **break** and **return** functions can be used to terminate the loop prematurely. The return value of this function is FALSE unless the **return** function is used to terminate the loop. Variables from an outer scope may be used within the loop, but the loop variable (if specified) masks any outer variables of the same name. Loops can be nested.

Example

```
CLIPS> (loop-for-count 2 (printout t "Hello world" crlf))
Hello world
Hello world
FALSE
CLIPS>
(loop-for-count (?cnt1 2 4) do
   (loop-for-count (?cnt2 1 3) do
         (println ?cnt1 " " ?cnt2)))
2 1
```

```
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
FALSE
CLIPS>
```

12.6.5 Progn

The **progn** function evaluates all of its arguments and returns the value of the last argument.

Syntax

```
(progn <expression>*)
```

Example

```
CLIPS> (progn (setgen 5) (gensym))
gen5
CLIPS>
```

12.6.6 Progn$

The **progn$** function performs a set of actions for each field of a multifield value. The field of the current iteration can be examined with <field-variable>, if specified. The index of the current iteration can be examined with <field-variable>**-index**. The **progn$** function can use variables from outer scopes, and the **return** and **break** functions can also be used within a **progn$** as long as they are valid in the outer scope. The return value of this function is the return value of the last action performed for the last field in the multifield value.

Syntax

```
(progn$ <multifield-spec> <expression>*)

<multifield-spec> ::= <multifield-expression> |
                      (<field-variable> <multifield-expression>)
```

Example

```
CLIPS> (progn$ (?field (create$ abc def ghi))
          (println "--> " ?field " " ?field-index " <--"))
--> abc 1 <--
--> def 2 <--
--> ghi 3 <--
CLIPS>
```

12.6.7 Return

The **return** function immediately terminates the currently executing deffunction, generic function method, message-handler, defrule RHS, or certain instance set query functions (**do-for-instance**, **do-for-all-instances** and **delayed-do-for-all-instances**). Without any arguments, there is no return value. However, if an argument is included, its evaluation is given as the return value of the deffunction , method or message-handler.

The **return** function can only be used within the actions of deffunctions, methods and message-handlers, defrules, or the instance set query functions previously listed. If used on the RHS of a rule, the current focus is removed from the focus stack. In addition, **return** should not be used as an argument to another function call. If used within an instance set query function, the **return** function is only valid if it is applicable in the outer scope of the query.

Syntax

```
(return [<expression>])
```

Example

```
CLIPS>
(deffunction sign (?num)
  (if (> ?num 0) then
     (return 1))
  (if (< ?num 0) then
     (return -1))
  0)
CLIPS> (sign 5)
1
CLIPS> (sign -10)
-1
CLIPS> (sign 0)
0
CLIPS>
```

12.6.8 Break

The **break** function immediately terminates the currently iterating **while** loop, **loop-for-count** execution, **progn** execution, **progn$** execution, **foreach** execution, or certain instance set query functions (**do-for-instance**, **do-for-all-instances** and **delayed-do-for-all-instances**).

The **break** function can only be used within the actions of a **while** loop, **loop-for-count** execution, **progn** execution, **progn$** execution, **foreach** execution, or the specified instance set queries previously listed. Other uses will have no effect. The **break** cannot be used within a **progn** unless it is valid for the outer scope of the **progn**. In addition, **break** should not be used as an argument to another function call.

Syntax

```
(break)
```

Example

```
CLIPS>
(deffunction iterate (?num)
  (bind ?i 0)
  (while TRUE do
    (if (>= ?i ?num) then
       (break))
    (print ?i " ")
    (bind ?i (+ ?i 1)))
  (println))
CLIPS> (iterate 1)
0
CLIPS> (iterate 10)
0 1 2 3 4 5 6 7 8 9
CLIPS>
```

12.6.9 Switch

The **switch** function allows a particular group of actions (among several groups of actions) to be performed based on a specified value.

Syntax

```
(switch <test-expression>
   <case-statement>*
   [<default-statement>])

<case-statement> ::=
   (case <comparison-expression> then <action>*)

<default-statement> ::= (default <action>*)
```

As indicated by the BNF, the optional default statement must succeed all case statements. None of the case comparison expressions should be the same.

The **switch** function evaluates the <test-expression> first and then evaluates each <comparison-expression> in order of definition. Once the evaluation of the <comparison-expression> is equivalent to the evaluation of the <test-expression>, the actions of that case are evaluated (in order) and the switch function is terminated. If no cases are satisfied, the default actions (if any) are evaluated (in order).

The return value of the **switch** function is the last action evaluated in the **switch** function. If no actions are evaluated, the return value is the symbol FALSE.

Example

```
CLIPS> (defglobal ?*x* = 0)
CLIPS> (defglobal ?*y* = 1)
CLIPS>
(deffunction foo (?val)
   (switch ?val
      (case ?*x* then *x*)
      (case ?*y* then *y*)
      (default none)))
CLIPS> (foo 0)
*x*
CLIPS> (foo 1)
*y*
CLIPS> (foo 2)
none
CLIPS>
```

12.6.10 Foreach

The **foreach** function performs a set of actions for each field of a multifield value. The field of the current iteration can be examined with <field-variable>, if specified. The index of the current iteration can be examined with <field-variable>**-index**. The **foreach** function can use variables from outer scopes, and the **return** and **break** functions can also be used within a **foreach** as long as they are valid in the outer scope. The return value of this function is the return value of the last action performed for the last field in the multifield value.

Syntax

```
(foreach <field-variable> <multifield-expression> <expression>*)
```

Example

```
CLIPS> (foreach ?field (create$ abc def ghi)
          (println "--> " ?field " " ?field-index " <--"))
--> abc 1 <--
--> def 2 <--
--> ghi 3 <--
CLIPS>
```

❖ Portability Note

The **foreach** function provides the same functionality as the **progn$** function, but uses different syntax with a more meaningful function name. It is provided for compatibility with Jess (Java Expert System Shell).

12.7 Miscellaneous Functions

The following are additional functions for use within CLIPS.

12.7.1 Gensym

The **gensym** function returns a special, sequenced symbol that can be stored as a single field. This is primarily for tagging patterns that need a unique identifier, but the user does not care what the identifier is. Multiple calls to **gensym** are guaranteed to return different identifiers of the form

```
genX
```

where **X** is a positive integer. The first call to **gensym** returns **gen1**; all subsequent calls increment the number. Note that **gensym** is *not* reset after a call to **clear**. If users plan to use the gensym feature, they should avoid creating facts which include a user-defined field of this form.

Example

```
(assert (new-id (gensym) flag1 7))
```

which, on the first call, generates a fact of the form

```
(new-id gen1 flag1 7)
```

12.7.2 Gensym*

The **gensym*** function is similar to the **gensym** function, however, it will produce a unique symbol that does not currently exist within the CLIPS environment.

Example

```
CLIPS> (setgen 1)
1
CLIPS> (assert (gen1 gen2 gen3))
<Fact-1>
CLIPS> (gensym)
gen1
CLIPS> (gensym*)
gen4
CLIPS>
```

12.7.3 Setgen

The **setgen** function allows the user to set the starting number used by the **gensym** and **gensym*** functions.

Syntax

```
(setgen <integer-expression>)
```

where <intger-expression> must be a positive integer value and is the value returned by this function. All subsequent calls to **gensym** will return a sequenced symbol with the numeric portion of the symbol starting at <integer-expression>.

Example

```
CLIPS> (setgen 32)
32
CLIPS>
```

After this, calls to **gensym** will return gen32, gen33, etc.

12.7.4 Random

The **random** function returns a "random" integer value. It is patterned after the ANSI standard rand library function and therefore may not be available on all platforms.

Syntax

```
(random [<start-integer-expression> <end-integer-expression>])
```

where <start-integer-expression> and <end-integer-expression> if specified indicate the range of values to which the randomly generated integer is limited.

Example

```
(defrule roll-the-dice
   (roll-the-dice)
   =>
   (bind ?roll1 (random 1 6))
   (bind ?roll2 (random 1 6))
   (println "Your roll is:  " ?roll1 " " ?roll2))
```

12.7.5 Seed

The **seed** function seeds the random number generator. It is patterned after the ANSI standard seed library function and therefore may not be available on all platforms.

Syntax

```
(seed <integer-expression>)
```

where <integer-expression> is the integer seed value and the function has no return value.

12.7.6 Time

The **time** function returns a floating-point value representing the elapsed seconds since the system reference time.

Syntax

```
(time)
```

12.7.7 Determining the Restrictions for a Function

The **get-function-restrictions** function can be used to gain access to the restriction string associated with a CLIPS or user defined function. The restriction string contains information on the number and types of arguments that a function expects. See section 3.1 of the Advanced Programming Guide for the meaning of the characters which compose the restriction string.

Syntax

```
(get-function-restrictions <function-name>)
```

Example

```
CLIPS> (get-function-restrictions +)
"2*n"
CLIPS>
```

12.7.8 Sorting a List of Values

The function **sort** allows a list of values to be sorted based on a user specified comparison function.

Syntax

```
(sort <comparison-function-name> <expression>*)
```

This function returns a multifield value containing the sorted values specified as arguments. The comparison function used for sorting should accept exactly two arguments and can be a user-defined function, a generic function, or a deffunction. Given two adjacent arguments from the list to be sorted, the comparison function should return TRUE if its first argument should come after its second argument in the sorted list.

Example

```
CLIPS> (sort > 4 3 5 7 2 7)
(2 3 4 5 7 7)
CLIPS>
```

```
(deffunction string> (?a ?b)
  (> (str-compare ?a ?b) 0))
CLIPS> (sort string> ax aa bk mn ft m)
(aa ax bk ft m mn)
CLIPS>
```

12.7.9 Calling a Function

The function **funcall** constructs a function call from its arguments and then evaluates the function call. The first argument should be the name of a user-defined function, deffunction, or generic function. The remaining arguments are evaluated and then passed to the specified function when it is evaluated. Functions that are invoked using specialized syntax, such as the **assert** command (which uses parentheses to delimit both slot and function names), may not be called using funcall.

Syntax

```
(funcall <function-name> <expression>*)
```

Example

```
CLIPS> (funcall delete$ (create$ a b c) 2 2)
(a c)
CLIPS> (funcall + 1 2 (expand$ (create$ 3 4)))
10
CLIPS>
```

12.7.10 Timing Functions and Commands

The function **timer** returns the number of seconds elapsed evaluating a series of expressions.

Syntax

```
(timer <expression>*)
```

Example

```
CLIPS> (timer (loop-for-count 10000 (+ 3 4)))
0.0416709999999512
CLIPS>
```

12.7.11 Determining the Operating System

The **operating-system** function returns a symbol indicating the opertating system on which CLIPS is running. Possible return values are UNIX-V, UNIX-7, LINUX, DARWIN, MAC-OS-X, DOS, WINDOWS, and UNKNOWN.

Syntax

```
(operating-system)
```

12.7.12 Local Time

The **local-time** function returns a multifield value containing fields indicating the local time. In order, the returned fields are year, month, day, hours, seconds, day of week, days since the beginning of the year, and a boolean flag indicating whether daylight savings time is in effect.

Syntax

```
(local-time)
```

Example

```
CLIPS> (local-time)
(2017 3 12 19 45 47 Sunday 70 TRUE)
CLIPS>
```

12.7.13 Greenwich Mean Time

The **gm-time** function returns returns a multifield value containing fields indicating Greenwich mean time. In order, the returned fields are year, month, day, hours, seconds, day of week, days since the beginning of the year, and a boolean flag indicating whether daylight savings time is in effect.

Syntax

```
(gm-time)
```

Example

```
CLIPS> (gm-time)
(2017 3 13 0 45 49 Monday 71 FALSE)
CLIPS>
```

12.7.14 Getting the Error State

The **get-error** function returns returns the current value of the error state. The error state can be set by system and user defined functions as well as using the **set-error** function.

Syntax

```
(get-error)
```

12.7.15 Clearing the Error State

The **clear-error** function resets the error state to the value FALSE and returns the prior value of the error state.

Syntax

```
(clear-error)
```

12.7.16 Setting the Error State

The **set-error** function sets the error state.

Syntax

```
(set-error <expression>)
```

Example

```
CLIPS> (set-error 10)
CLIPS> (get-error)
10
CLIPS> (get-error)
10
CLIPS> (clear-error)
10
CLIPS> (get-error)
FALSE
CLIPS>
```

12.8 Deftemplate Functions

The following functions provide ancillary capabilities for the deftemplate construct.

12.8.1 Determining the Module in which a Deftemplate is Defined

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

Syntax

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

12.8.2 Getting the Allowed Values for a Deftemplate Slot

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

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(deftemplate A
  (slot x)
  (slot y (allowed-integers 2 3) (allowed-symbols foo)))
CLIPS> (deftemplate-slot-allowed-values A x)
FALSE
CLIPS> (deftemplate-slot-allowed-values A y)
(2 3 foo)
CLIPS>
```

12.8.3 Getting the Cardinality for a Deftemplate Slot

This function groups the minimum and maximum cardinality allowed for a multifield slot into a multifield variable. A maximum cardinality of infinity is indicated by the symbol **+oo** (the plus character followed by two lowercase o's—not zeroes). A multifield of length zero is returned for single field slots or if an error occurs.

Syntax

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

Example

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

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>
```

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>
```

12.8.8 Determining the Slot Names Associated with a Deftemplate

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

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (deftemplate foo (slot bar) (multislot yak))
CLIPS> (deftemplate-slot-names foo)
(bar yak)
CLIPS>
```

12.8.9 Getting the Numeric Range for a Deftemplate 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

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

Example

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

12.8.10 Testing whether a Deftemplate Slot is a Single-Field Slot

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

Syntax

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

Example

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

12.8.11 Getting the Primitive Types for a Deftemplate Slot

This function groups the names of the primitive types allowed for a deftemplate slot into a multifield variable. A multifield of length zero is returned if an error occurs.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (deftemplate A (slot y (type INTEGER LEXEME)))
CLIPS> (deftemplate-slot-types A y)
(INTEGER SYMBOL STRING)
CLIPS>
```

12.8.12 Getting the List of Deftemplates

The function **get-deftemplate-list** returns a multifield value containing the names of all deftemplate constructs facts 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 deftemplates are returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (deftemplate A)
CLIPS> (deftemplate B)
CLIPS> (get-deftemplate-list)
(A B)
CLIPS>
```

12.9 Fact Functions

The following actions are used for assert, retracting, and modifying facts.

12.9.1 Creating New Facts

The **assert** action allows the user to add a fact to the fact-list. Multiple facts may be asserted with each call. If the facts item is being watched (see section 13.2.3), then an informational message will be printed each time a fact is asserted.

Syntax

```
(assert <RHS-pattern>+)
```

Missing slots in a template fact being asserted are assigned their default value (see section 3). If an identical copy of the fact already exists in the fact-list, the fact will not be added (however, this behavior can be changed, see sections 13.4.4 and 13.4.5). Note that in addition to constants, expressions can be placed within a fact to be asserted. The first field of a fact must be a symbol. The value returned of the assert function is the fact-address of the last fact asserted. If an identical copy of the last fact already exists in the fact-list, then the fact-address of the existing fact is returned. If the assertion of the last fact causes an error, then the symbol FALSE is returned.

Example

```
CLIPS> (clear)
CLIPS> (assert (color red))
<Fact-1>
CLIPS> (assert (color blue) (value (+ 3 4)))
<Fact-3>
CLIPS> (assert (color red))
FALSE
CLIPS> (deftemplate status (slot temp) (slot pressure))
CLIPS> (assert (status (temp high)
                       (pressure low)))
<Fact-4>
CLIPS> (facts)
f-1     (color red)
f-2     (color blue)
f-3     (value 7)
f-4     (status (temp high) (pressure low))
For a total of 4 facts.
CLIPS>
```

12.9.2 Removing Facts from the Fact-list

The **retract** action allows the user to remove facts from the fact-list. Multiple facts may be retracted with a single retract statement. The retraction of a fact also removes all rules that depended upon that fact for activation from the agenda. Retraction of a fact may also cause the retraction of other facts which receive logical support from the retracted fact. If the facts item is being watched (see section 13.2.3), then an informational message will be printed each time a fact is retracted.

Syntax

```
(retract <retract-specifier>+ | *)

<retract-specifier> ::= <fact-specifier> | <integer-expression>
```

The term <retract-specifier> includes variables bound on the LHS to fact-addresses as described in section 5.4.1.8, or the **fact-index** of the desired fact (e.g. 3 for the fact labeled f-3), or an expression which evaluates to a retract-specifier. If the symbol * is used as an argument, all facts will be retracted. Note that the number generally is not known during the execution of a program, so facts usually are retracted by binding them on the LHS of a rule. Only variables, fact indices, or the symbol * may be used in a retract. External functions may *not* be called. This function has no return value.

Example

```
(defrule change-valve-status
   ?f1 <- (valve (id ?v) (state open))
   ?f2 <- (set (id ?v) (state close))
   =>
   (retract ?f2)
   (modify ?f1 (state close)))
```

12.9.3 Modifying Template Facts

The **modify** action allows the user to modify template facts on the fact-list. Only one fact may be modified with a single modify statement. The modification of a fact is equivalent to retracting the present fact and asserting the modified fact. Therefore, any facts receiving logical support from a template fact are retracted (assuming no other support) when the template fact is modified and the new template fact loses any logical support that it previously had.

Syntax

```
(modify <fact-specifier> <RHS-slot>*)
```

The term <fact-specifier> includes variables bound on the LHS to fact-addresses as described in section 5.4.1.8 or the fact-index of the desired fact (e.g. 3 for the fact labeled f-3). Note that the fact-index generally is not known during the execution of a program, so facts usually are modified by binding them on the LHS of a rule. Static deftemplate checking is not performed when a fact-index is used as the <fact-specifier> since the deftemplate being referenced is usually ambiguous. The value returned by this function is the fact-address of the newly modified fact. If an identical copy of the newly modified fact already exists in the fact-list, the fact-address of the existing copy is returned; otherwise the fact-address and fact-index of a modified fact is preserved. If the assertion of the newly modified fact causes an error, the symbol FALSE is returned. If all slot changes specified in the modify command match the current values of the fact to be modified, no action is taken.

Example

```
(defrule change-valve-status
   ?f1 <- (valve (id ?v) (state open))
   ?f2 <- (set (id ?v) (state close))
   =>
   (retract ?f2)
   (modify ?f1 (state close)))
```

12.9.4 Duplicating Template Facts

The **duplicate** action allows the user to duplicate deftemplate facts on the fact-list changing a group of specified fields. This command allows a new fact to be created by copying most of the fields of a source fact and then specifying the fields to be changed. Only one fact may be duplicated with a single duplicate statement. The duplicate command is similar to the modify command except the fact being duplicated is not retracted.

Syntax

```
(duplicate <fact-specifier> <RHS-slot>*)
```

The term <fact-specifier> includes variables bound on the LHS to fact-addresses as described in section 5.4.1.8 or the fact-index of the desired fact (e.g. 3 for the fact labeled f-3). Note that the fact-index generally is not known during the execution of a program, so facts usually are duplicated by binding them on the LHS of a rule. Static deftemplate checking is not performed when a fact-index is used as the <fact-specifier> since the deftemplate being referenced is usually ambiguous. The value returned by this function is the fact-address of the newly duplicated fact. If an identical copy of the newly duplicated fact already exists in the fact-list, the fact-address of the existing copy is returned. If the assertion of the newly duplicated fact causes an error, then the symbol FALSE is returned.

Example

```
(defrule duplicate-part
   ?f1 <- (duplicate-part (id ?name))
   ?f2 <- (part (id ?name))
   =>
   (retract ?f1)
   (duplicate ?f2 (id (gensym*))))
```

12.9.5 Asserting a String

The **assert-string** function is similar to assert in that it will add a fact to the fact-list. However, **assert-string** takes a single string representing a fact (expressed in either ordered or deftemplate format) and asserts it. Only one fact may be asserted with each **assert-string** statement.

Syntax

```
(assert-string <string-expression>)
```

If an identical copy of the fact already exists in the fact-list, the fact will not be added (however, this behavior can be changed, see sections 13.4.4 and 13.4.5). Fields within the fact may contain a string by escaping the quote character with a backslash. Note that this function takes a string and turns it into fields. If the fields within that string are going to contain special characters (such as a backslash), they need to be escaped twice (because you are literally embedding a string within a string and the backslash mechanism ends up being applied twice). Global variables and expressions can be contained within the string. The value returned by this function is the fact-address of the newly created fact. If an identical copy of the newly created fact already exists in the fact-list, the fact-address of the existing copy is returned. If the assertion of the newly created fact causes an error, then the symbol FALSE is returned.

Example

```
CLIPS> (clear)
CLIPS> (deftemplate foo (slot x) (slot y))
CLIPS> (assert-string "(status valve open)")
<Fact-1>
CLIPS> (assert-string "(light \"red\")")
<Fact-2>
CLIPS> (assert-string "(a\\b \"c\\\\d\")")
<Fact-3>
CLIPS> (assert-string "(foo (x 3))")
<Fact-4>
CLIPS> (assert-string "(foo (y 7))")
<Fact-5>
CLIPS> (facts)
f-1     (status valve open)
f-2     (light "red")
f-3     (a\b "c\d")
f-4     (foo (x 3) (y nil))
f-5     (foo (x nil) (y 7))
For a total of 5 facts.
CLIPS>
```

12.9.6 Getting the Fact-Index of a Fact-address

The **fact-index** function returns the fact-index (an integer) of a fact-address.

Syntax

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

Example

```
(defrule print-fact-indices
   ?f <- (some-fact $?)
   =>
   (println (fact-index ?f)))
```

12.9.7 Determining If a Fact Exists

The **fact-existp** returns TRUE if the fact specified by its fact-index or fact-address arguments exists, otherwise FALSE is returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (defglobal ?*x* = (assert (example fact)))
CLIPS> (facts)
f-1     (example fact)
For a total of 1 fact.
CLIPS> (fact-existp 1)
TRUE
CLIPS> (retract 1)
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-1>
CLIPS> (fact-relation 1)
example
CLIPS>
```

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-1>
CLIPS> (fact-slot-names 1)
(bar yak)
CLIPS> (assert (another a b c))
<Fact-2>
CLIPS> (fact-slot-names 2)
(implied)
CLIPS>
```

12.9.10 Retrieving the Slot Value of a Fact

The **fact-slot-value** function returns the value of the specified slot from the specified fact. The symbol *implied* should be used as the slot name for the implied multifield slot of an ordered fact. FALSE is returned if the slot name argument is invalid or the specified fact does not exist.

Syntax

```
(fact-slot-value <fact-address-or-index> <slot-name>)
```

Example

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

12.9.11 Retrieving the Fact-List

The **get-fact-list** function returns a multifield containing the list of facts 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 facts are returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (assert (a))
<Fact-1>
CLIPS> (get-fact-list)
(<Fact-1>)
CLIPS> (defmodule B)
CLIPS> (assert (b))
<Fact-2>
CLIPS> (get-fact-list)
(<Fact-2>)
CLIPS> (get-fact-list MAIN)
(<Fact-1>)
CLIPS> (get-fact-list *)
(<Fact-1> <Fact-2>)
CLIPS>
```

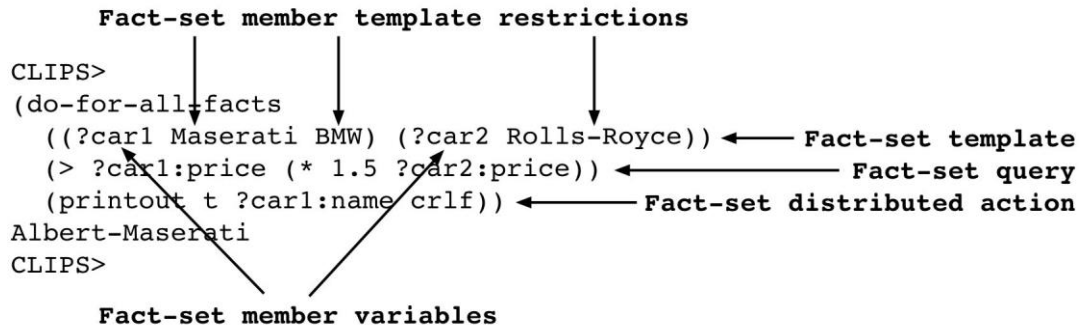
12.9.12 Fact-set Queries and Distributed Actions

CLIPS provides a useful query system for determining and performing actions on sets of facts that satisfy user-defined queries. The fact query system in CLIPS provides six functions, each of which operate on fact-sets determined by user-defined criteria:

Function	Purpose
any-factp	Determines if one or more fact-sets satisfy a query
find-fact	Returns the first fact-set that satisfies a query
find-all-facts	Groups and returns all fact-sets which satisfy a query
do-for-fact	Performs an action for the first fact-set which satisfies a query
do-for-all-facts	Performs an action for every fact-set which satisfies a query as they are found
delayed-do-for-all-facts	Groups all fact-sets which satisfy a query and then iterates an action over this group

Explanations on how to form fact-set templates, queries and actions immediately follow, for these definitions are common to all of the query functions. The specific details of each query function will then be given. The following is a complete example of a fact-set query function:

Example



For all of the examples in this section, assume that the commands below have already been entered:

Example

```
CLIPS> (clear)
CLIPS>
 (deftemplate girl
  (slot name)
  (slot sex (default female))
  (slot age (default 4)))
CLIPS>
(deftemplate woman
  (slot name)
  (slot sex (default female))
  (slot age (default 25)))
CLIPS>
(deftemplate boy
  (slot name)
  (slot sex (default male))
  (slot age (default 4)))
CLIPS>
(deftemplate man
  (slot name)
  (slot sex (default male))
  (slot age (default 25)))
CLIPS>
(deffacts PEOPLE
  (man (name Man-1) (age 18))
  (man (name Man-2) (age 60))
  (woman (name Woman-1) (age 18))
  (woman (name Woman-2) (age 60))
  (woman (name Woman-3))
  (boy (name Boy-1) (age 8))
  (boy (name Boy-2))
  (boy (name Boy-3))
  (boy (name Boy-4))
  (girl (name Girl-1) (age 8))
  (girl (name Girl-2)))
CLIPS> (reset)
CLIPS> (facts)
f-1     (man (name Man-1) (sex male) (age 18))
f-2     (man (name Man-2) (sex male) (age 60))
f-3     (woman (name Woman-1) (sex female) (age 18))
f-4     (woman (name Woman-2) (sex female) (age 60))
f-5     (woman (name Woman-3) (sex female) (age 25))
f-6     (boy (name Boy-1) (sex male) (age 8))
```

```
f-7     (boy (name Boy-2) (sex male) (age 4))
f-8     (boy (name Boy-3) (sex male) (age 4))
f-9     (boy (name Boy-4) (sex male) (age 4))
f-10    (girl (name Girl-1) (sex female) (age 8))
f-11    (girl (name Girl-2) (sex female) (age 4))
For a total of 11 facts.
CLIPS>
```

12.9.12.1 Fact-set Definition

A **fact-set** is an ordered collection of facts. Each **fact-set member** is a member of a set of deftemplates, called **template restrictions**, defined by the user. The template restrictions can be different for each fact-set member. The query functions use **fact-set templates** to generate fact-sets. A fact-set template is a set of **fact-set member variables** and their associated template restrictions. Fact-set member variables reference the corresponding members in each fact-set which matches a template. Variables may be used to specify the deftemplates for the fact-set template, but if the constant names of the deftemplates are specified, the deftemplates must already be defined. Module specifiers may be included with the deftemplate names; the deftemplates need not be in scope of the current module.

Syntax

```
<fact-set-template>
      ::= (<fact-set-member-template>+)
<fact-set-member-template>
      ::= (<fact-set-member-variable> <deftemplate-restrictions>)
<fact-set-member-variable>   ::= <single-field-variable>
<deftemplate-restrictions>   ::= <deftemplate-name-expression>+
```

Example

One fact-set template might be the ordered pairs of boys or men and girls or women.

```
((?man-or-boy boy man) (?woman-or-girl girl woman))
```

Fact-set member variables (e.g. ?man-or-boy) are bound to fact-addresses.

12.9.12.2 Fact-set Determination

CLIPS uses straightforward permutations to generate fact-sets that match a fact-set template from the actual facts in the system. The rules are as follows:

1) When there is more than one member in a fact-set template, vary the rightmost members first.

2) When there is more than one deftemplate that an fact-set member can be, iterate through the deftemplate from left to right.

3) Examine facts of a deftemplate in the order that they were defined.

Example

For the fact-set template given in section 12.9.12.1, thirty fact-sets would be generated in the following order:

```
1.  <Fact-6> <Fact-10>
2.  <Fact-6> <Fact-11>
3.  <Fact-6> <Fact-3>
4.  <Fact-6> <Fact-4>
5.  <Fact-6> <Fact-5>
6.  <Fact-7> <Fact-10>
7.  <Fact-7> <Fact-11>
8.  <Fact-7> <Fact-3>
9.  <Fact-7> <Fact-4>
10. <Fact-7> <Fact-5>
11. <Fact-8> <Fact-10>
12. <Fact-8> <Fact-11>
13  <Fact-8> <Fact-3>
14. <Fact-8> <Fact-4>
15. <Fact-8> <Fact-5>
16. <Fact-9> <Fact-10>
17. <Fact-9> <Fact-11>
18. <Fact-9> <Fact-3>
19. <Fact-9> <Fact-4>
20. <Fact-9> <Fact-5>
21. <Fact-1> <Fact-10>
22. <Fact-1> <Fact-11>
23. <Fact-1> <Fact-3>
24. <Fact-1> <Fact-4>
25. <Fact-1> <Fact-5>
26. <Fact-2> <Fact-10>
27. <Fact-2> <Fact-11>
28. <Fact-2> <Fact-3>
29. <Fact-2> <Fact-4>
30. <Fact-2> <Fact-5>
```

12.9.12.3 Query Definition

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

Syntax

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

Example

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

```
(= (fact-slot-value ?man-or-boy age) (fact-slot-value ?woman-or-girl age))
```

Within a query, slots of fact-set members can be directly read with a shorthand notation similar to that used by instances in message-handlers (see section 9.4.2).

Syntax

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

Example

The previous example could be rewritten as:

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

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

12.9.12.4 Distributed Action Definition

A **distributed action** is a user-defined expression evaluated for each fact-set which satisfies a query.

Action Syntax

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

Example

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

```
(println "(" ?man-or-boy:name "," ?woman-or-girl:name ")")
```

12.9.12.5 Scope in Fact-set Query Functions

A fact-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 a fact-set member variable, then that variable may be referenced within the query and action. In addition, rebinding variables within a fact-set function action is allowed. However, attempts to rebind fact-set member variables will generate errors. Binding variables is not allowed within a query. Fact-set query functions can be nested.

Example

```
CLIPS>
(deffunction count-facts (?template)
  (bind ?count 0)
  (do-for-all-facts ((?fct ?template)) TRUE
    (bind ?count (+ ?count 1)))
 ?count)
CLIPS>
(deffunction count-facts-2 (?template)
  (length (find-all-facts ((?fct ?template)) TRUE)))
CLIPS> (count-facts woman)
3
CLIPS> (count-facts-2 boy)
4
```

```
CLIPS>
```

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

Example

```
CLIPS>
(deffunction last-fact (?template)
   (any-factp ((?fct ?template)) TRUE)
   ?fct)

[PRCCODE3] Undefined variable fct referenced in deffunction.

ERROR:
(deffunction MAIN::last-fact
   (?template)
   (any-factp ((?fct ?template))
      TRUE)
   ?fct
   )
CLIPS>
```

12.9.12.6 Errors during Fact-set Query Functions

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

12.9.12.7 Halting and Returning Values from Query Functions

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

12.9.12.8 Fact-set Query Functions

The fact query system in CLIPS provides six functions. For a given set of facts, all six query functions will iterate over these facts in the same order (see section 12.9.12.2). However, if a particular fact is retracted and reasserted, the iteration order will change.

12.9.12.8.1 Testing if Any Fact-set Satisfies a Query

This function applies a query to each fact-set which matches the template. If a fact-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-factp <fact-set-template> <query>)
```

Example

Are there any men over age 30?

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

12.9.12.8.2 Determining the First Fact-set Satisfying a Query

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

Syntax

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

Example

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

```
CLIPS>
(find-fact((?m man) (?w woman)) (= ?m:age ?w:age))
(<Fact-1> <Fact-3>)
CLIPS>
```

12.9.12.8.3 Determining All Fact-sets Satisfying a Query

This function applies a query to each fact-set which matches the template. Each fact-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 fact-sets. If there are n facts in each fact-set, and m fact-sets satisfied the query, then the length of the returned multifield value will be n * m. The first n fields correspond to the first fact-set, and so on. Each field of the multifield value is an fact-address representing a fact-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-facts <fact-set-template> <query>)
```

Example

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

```
CLIPS>
(find-all-facts ((?m man) (?w woman)) (= ?m:age ?w:age))
(<Fact-1> <Fact-3> <Fact-2> <Fact-4>)
CLIPS>
```

12.9.12.8.4 Executing an Action for the First Fact-set Satisfying a Query

This function applies a query to each fact-set which matches the template. If a fact-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 fact-set satisfied the query, then the return value is the symbol FALSE.

Syntax

```
(do-for-fact <fact-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-fact ((?p1 girl boy woman man)
              (?p2 girl boy woman man)
              (?p3 girl boy woman man))
  (and (= ?p1:age ?p2:age ?p3:age)
       (neq ?p1 ?p2)
       (neq ?p1 ?p3)
       (neq ?p2 ?p3))
  (println ?p1:name " " ?p2:name " " ?p3:name))
Girl-2 Boy-2 Boy-3
CLIPS>
```

12.9.12.8.5 Executing an Action for All Fact-sets Satisfying a Query

This function applies a query to each fact-set which matches the template. If a fact-set satisfies the query, the specified action is executed. The return value is the evaluation of the action for the last fact-set which satisfied the query. If no fact-set satisfied the query, then the return value is the symbol FALSE.

Syntax

```
(do-for-all-facts <fact-set-template> <query> <action>*)
```

Example

Print out all triplets of different people that have the same age. The calls to **str-compare** limit the fact-sets which satisfy the query to combinations instead of permutations. Without these restrictions, two fact-sets which differed only in the order of their members would both satisfy the query.

```
CLIPS>
(do-for-all-facts ((?p1 girl boy woman man)
                   (?p2 girl boy woman man)
                   (?p3 girl boy woman man))
  (and (= ?p1:age ?p2:age ?p3:age)
       (> (str-compare ?p1:name ?p2:name) 0)
       (> (str-compare ?p2:name ?p3:name) 0))
  (println ?p1:name " " ?p2:name " " ?p3:name))
Girl-2 Boy-3 Boy-2
Girl-2 Boy-4 Boy-2
Girl-2 Boy-4 Boy-3
Boy-4 Boy-3 Boy-2
CLIPS>
```

12.9.12.8.6 Executing a Delayed Action for All Fact-sets Satisfying a Query

This function is similar to **do-for-all-facts** except that it groups all fact-sets which satisfy the query into an intermediary multifield value. If there are no fact-sets which satisfy the query, then the function returns the symbol FALSE. Otherwise, the specified action is executed for each fact-set in the multifield value, and the return value is the evaluation of the action for the last fact-set to satisfy the query. The intermediary multifield value is discarded. This function can consume large amounts of memory in the same fashion as **find-all-facts**. This function should be used in lieu of **do-for-all-facts** when the action applied to one fact-set would change the result of the query for another fact-set (unless that is the desired effect).

Syntax

```
(delayed-do-for-all-facts <fact-set-template>
  <query> <action>*)
```

Example

Delete all boys with the greatest age. The test in this case is another query function which determines if there are any older boys than the one currently being examined. The action needs to be delayed until all boys have been processed, or the greatest age will decrease as the older boys are deleted.

```
CLIPS> (watch facts)
CLIPS>
(delayed-do-for-all-facts ((?b1 boy))
  (not (any-factp ((?b2 boy)) (> ?b2:age ?b1:age)))
  (retract ?b1))
```

```
<== f-6     (boy (name Boy-1) (sex male) (age 8))
CLIPS> (unwatch facts)
CLIPS> (reset)
CLIPS> (watch facts)
CLIPS>
(do-for-all-facts ((?b1 boy))
  (not (any-factp ((?b2 boy)) (> ?b2:age ?b1:age)))
  (retract ?b1))
<== f-6     (boy (name Boy-1) (sex male) (age 8))
<== f-7     (boy (name Boy-2) (sex male) (age 4))
<== f-8     (boy (name Boy-3) (sex male) (age 4))
<== f-9     (boy (name Boy-4) (sex male) (age 4))
CLIPS> (unwatch facts)
CLIPS>
```

12.10 Deffacts Functions

The following functions provide ancillary capabilities for the deffacts construct.

12.10.1 Getting the List of Deffacts

The function **get-deffacts-list** returns a multifield value containing the names of all deffacts 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 deffacts are returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (get-deffacts-list)
()
CLIPS> (deffacts foo)
CLIPS> (get-deffacts-list)
(foo)
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>
```

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.

12.12.1 Getting the Current Focus

The function **get-focus** returns the module name of the current focus. If the focus stack is empty, then the symbol FALSE is returned.

Syntax

```
(get-focus)
```

Example

```
CLIPS> (clear)
CLIPS> (get-focus)
MAIN
CLIPS> (defmodule A)
```

```
CLIPS> (defmodule B)
CLIPS> (focus A B)
TRUE
CLIPS> (get-focus)
A
CLIPS>
```

12.12.2 Getting the Focus Stack

The function **get-focus-stack** returns all of the module names in the focus stack as a multifield value. A multifield value of length zero is returned if the focus stack is empty.

Syntax

```
(get-focus-stack)
```

Example

```
CLIPS> (clear)
CLIPS> (get-focus-stack)
(MAIN)
CLIPS> (clear-focus-stack)
CLIPS> (get-focus-stack)
()
CLIPS> (defmodule A)
CLIPS> (defmodule B)
CLIPS> (focus A B)
TRUE
CLIPS> (get-focus-stack)
(A B)
CLIPS>
```

12.12.3 Removing the Current Focus from the Focus Stack

The function **pop-focus** removes the current focus from the focus stack and returns the module name of the current focus. If the focus stack is empty, then the symbol FALSE is returned.

Syntax

```
(pop-focus)
```

Example

```
CLIPS> (clear)
CLIPS> (list-focus-stack)
MAIN
CLIPS> (pop-focus)
MAIN
CLIPS> (defmodule A)
CLIPS> (defmodule B)
CLIPS> (focus A B)
TRUE
CLIPS> (list-focus-stack)
A
```

```
B
MAIN
CLIPS> (pop-focus)
A
CLIPS> (list-focus-stack)
B
CLIPS>
```

12.13 Defglobal Functions

The following functions provide ancillary capabilities for the defglobal construct.

12.13.1 Getting the List of Defglobals

The function **get-defglobal-list** returns a multifield value containing the names of all global variables 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 globals are returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (get-defglobal-list)
()
CLIPS> (defglobal ?*x* = 3 ?*y* = 5)
CLIPS> (get-defglobal-list)
(x y)
CLIPS>
```

12.13.2 Determining the Module in which a Defglobal is Defined

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

Syntax

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

12.14 Deffunction Functions

The following functions provide ancillary capabilities for the deffunction construct.

12.14.1 Getting the List of Deffunctions

The function **get-deffunction-list** returns a multifield value containing the names of all deffunction 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 deffunctions are returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (get-deffunction-list)
()
CLIPS> (deffunction foo ())
CLIPS> (deffunction bar ())
CLIPS> (get-deffunction-list)
(foo bar)
CLIPS>
```

12.14.2 Determining the Module in which a Deffunction is Defined

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

Syntax

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

12.15 Generic Function Functions

The following functions provide ancillary capabilities for generic function methods.

12.15.1 Getting the List of Defgenerics

The function **get-defgeneric-list** returns a multifield value containing the names of all defgeneric constructs that are currently defined.

Syntax

```
(get-defgeneric-list)
```

Example

```
CLIPS> (clear)
CLIPS> (get-defgeneric-list)
()
```

```
CLIPS> (defgeneric foo)
CLIPS> (defgeneric bar)
CLIPS> (get-defgeneric-list)
(foo bar)
CLIPS>
```

12.15.2 Determining the Module in which a Generic Function is Defined

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

Syntax

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

12.15.3 Getting the List of Defmethods

The function **get-defmethod-list** returns a multifield value containing method name/indices pairs for all defmethod constructs that are currently defined. The optional generic-function name parameter restricts the methods return to only those of the specified generic function.

Syntax

```
(get-defmethod-list [<generic-function-name>])
```

Example

```
CLIPS> (clear)
CLIPS> (get-defmethod-list)
()
CLIPS> (defmethod foo ((?x STRING)))
CLIPS> (defmethod foo ((?x INTEGER)))
CLIPS> (defmethod bar ((?x STRING)))
CLIPS> (defmethod bar ((?x INTEGER)))
CLIPS> (get-defmethod-list)
(foo 1 foo 2 bar 1 bar 2)
CLIPS> (get-defmethod-list foo)
(foo 1 foo 2)
CLIPS>
```

12.15.4 Type Determination

The function **type** returns a symbol which is the name of the type (or class) of its of argument. This function is equivalent to the **class** function, but, unlike the **class** function, it is available even when COOL is not installed.

Syntax

```
(type <expression>)
```

Example

```
CLIPS> (type (+ 2 2))
INTEGER
CLIPS> (defclass CAR (is-a USER))
CLIPS> (make-instance Rolls-Royce of CAR)
[Rolls-Royce]
CLIPS> (type Rolls-Royce)
SYMBOL
CLIPS> (type [Rolls-Royce])
CAR
CLIPS>
```

12.15.5 Existence of Shadowed Methods

If called from a method for a generic function, the function **next-methodp** will return the symbol TRUE if there is another method shadowed (see section 8.5.3) by the current one. Otherwise, the function will return the symbol FALSE.

Syntax

```
(next-methodp)
```

12.15.6 Calling Shadowed Methods

If the conditions are such that the function **next-methodp** would return the symbol TRUE, then calling the function **call-next-method** will execute the shadowed (see section 8.5.3) method. Otherwise, a method execution error will occur (see section 8.5.4). 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 method. The shadowed method is passed the same arguments as the calling method.

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

Syntax

```
(call-next-method)
```

Example

```
CLIPS>
(defmethod describe ((?a INTEGER))
  (if (next-methodp) then
     (bind ?extension (str-cat " " (call-next-method)))
   else
     (bind ?extension ""))
  (str-cat "INTEGER" ?extension))
CLIPS> (describe 3)
"INTEGER"
```

```
CLIPS>
(defmethod describe ((?a NUMBER))
  "NUMBER")
CLIPS> (describe 3)
"INTEGER NUMBER"
CLIPS> (describe 3.0)
"NUMBER"
CLIPS>
```

12.15.7 Calling Shadowed Methods with Overrides

The function **override-next-method** is similar to **call-next-method**, except that new arguments can be provided. This allows one method to act as a wrapper for another and set up a special environment for the shadowed method. From the set of methods which are more general than the currently executing one, the most specific method which is applicable to the new arguments is executed. (In contrast, **call-next-method** calls the next most specific method which is applicable to the same arguments as the currently executing one received.) A recursive call to the generic function itself should be used in lieu of **override-next-method** if the most specific of all methods for the generic function which is applicable to the new arguments should be executed.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(defmethod + ((?a INTEGER) (?b INTEGER))
  (override-next-method (* ?a 2) (* ?b 3)))
CLIPS> (list-defmethods +)
+ #2  (INTEGER) (INTEGER)
+ #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
For a total of 2 methods.
CLIPS> (preview-generic + 1 2)
+ #2  (INTEGER) (INTEGER)
+ #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
CLIPS> (watch methods)
CLIPS> (+ 1 2)
MTH >> +:#2  ED:1 (1 2)
MTH >> +:#SYS1  ED:2 (2 6)
MTH << +:#SYS1  ED:2 (2 6)
MTH << +:#2  ED:1 (1 2)
8
CLIPS> (unwatch methods)
CLIPS>
```

12.15.8 Calling a Specific Method

The function **call-specific-method** allows the user to call a particular method of a generic function without regards to method precedence. This allows the user to bypass method

precedence when absolutely necessary. The method must be applicable to the arguments passed. Shadowed methods can still be called via **call-next-method** and **override-next-method**.

Syntax

```
(call-specific-method <generic-function> <method-index>
                      <expression>*)
```

Example

```
CLIPS> (clear)
CLIPS>
(defmethod + ((?a INTEGER) (?b INTEGER))
  (* (- ?a ?b) (- ?b ?a)))
CLIPS> (list-defmethods +)
+ #2  (INTEGER) (INTEGER)
+ #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
For a total of 2 methods.
CLIPS> (preview-generic + 1 2)
+ #2  (INTEGER) (INTEGER)
+ #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
CLIPS> (watch methods)
CLIPS> (+ 1 2)
MTH >> +:#2  ED:1 (1 2)
MTH << +:#2  ED:1 (1 2)
-1
CLIPS> (call-specific-method + 1 1 2)
MTH >> +:#SYS1  ED:1 (1 2)
MTH << +:#SYS1  ED:1 (1 2)
3
CLIPS> (unwatch methods)
CLIPS>
```

12.15.9 Getting the Restrictions of Defmethods

The function **get-method-restrictions** returns a multifield value containing information about the restrictions for the specified method using the following format:

```
<minimum-number-of-arguments>
<maximum-number-of-arguments> (can be -1 for wildcards)
<number-of-restrictions>
<multifield-index-of-first-restriction-info>
      .
      .
      .
<multifield-index-of-nth-restriction-info>
<first-restriction-query> (TRUE or FALSE)
<first-restriction-class-count>
<first-restriction-first-class>
      .
      .
      .
<first-restriction-nth-class>
      .
      .
      .
<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.

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)
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 FOO
BAR)
CLIPS>
```

12.16.1.2 Determining the Module in which a Defclass is Defined

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

Syntax

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

12.16.1.3 Determining if a Class Exists

This function returns the symbol TRUE if the specified class is defined, FALSE otherwise.

Syntax

```
(class-existp <class-name>)
```

12.16.1.4 Superclass Determination

This function returns the symbol TRUE if the first class is a superclass of the second class, FALSE otherwise.

Syntax

```
(superclassp <class1-name> <class2-name>)
```

12.16.1.5 Subclass Determination

This function returns the symbol TRUE if the first class is a subclass of the second class, FALSE otherwise.

Syntax

```
(subclassp <class1-name> <class2-name>)
```

12.16.1.6 Slot Existence

This function returns the symbol TRUE if the specified slot is present in the specified class, FALSE otherwise. If the *inherit* keyword is specified then the slot may be inherited, otherwise it must be directly defined in the specified class.

Syntax

```
(slot-existp <class> <slot> [inherit])
```

12.16.1.7 Testing whether a Slot is Writable

This function returns the symbol TRUE if the specified slot in the specified class is writable (see section 9.3.3.4). Otherwise, it returns the symbol FALSE. An error is generated if the specified class or slot does not exist.

Syntax

```
(slot-writablep <class-expression> <slot-name-expression>)
```

12.16.1.8 Testing whether a Slot is Initializable

This function returns the symbol TRUE if the specified slot in the specified class is initializable (see section 9.3.3.4). Otherwise, it returns the symbol FALSE. An error is generated if the specified class or slot does not exist.

Syntax

```
(slot-initablep <class-expression> <slot-name-expression>)
```

12.16.1.9 Testing whether a Slot is Public

This function returns the symbol TRUE if the specified slot in the specified class is public (see section 9.3.3.8). Otherwise, it returns the symbol FALSE. An error is generated if the specified class or slot does not exist.

Syntax

```
(slot-publicp <class-expression> <slot-name-expression>)
```

12.16.1.10 Testing whether a Slot can be Accessed Directly

This function returns the symbol TRUE if the specified slot in the specified class can be accessed directly (see section 9.4.2). Otherwise, it returns the symbol FALSE. An error is generated if the specified class or slot does not exist.

Syntax

```
(slot-direct-accessp <class-expression> <slot-name-expression>)
```

12.16.1.11 Message-handler Existence

This function returns the symbol TRUE if the specified message-handler is defined (directly only, not by inheritance) for the class, FALSE otherwise.

Syntax

Defaults are in ***bold italics***.

```
(message-handler-existp <class-name> <handler-name> [<handler-type>])
<handler-type> ::= around | before | primary | after
```

12.16.1.12 Determining if a Class can have Direct Instances

This function returns the symbol TRUE if the specified class is abstract, i.e. the class cannot have direct instances, FALSE otherwise.

Syntax

```
(class-abstractp <class-name>)
```

12.16.1.13 Determining if a Class can Satisfy Object Patterns

This function returns the symbol TRUE if the specified class is reactive, i.e. objects of the class can match object patterns, FALSE otherwise.

Syntax

```
(class-reactivep <class-name>)
```

12.16.1.14 Getting the List of Superclasses for a Class

This function groups the names of the direct superclasses of a class into a multifield variable. If the optional argument “inherit” is given, indirect superclasses are also included. A multifield of length zero is returned if an error occurs.

Syntax

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

Example

```
CLIPS> (class-superclasses INTEGER)
(NUMBER)
CLIPS> (class-superclasses INTEGER inherit)
(NUMBER PRIMITIVE OBJECT)
CLIPS>
```

12.16.1.15 Getting the List of Subclasses for a Class

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

Syntax

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

Example

```
CLIPS> (class-subclasses PRIMITIVE)
(NUMBER LEXEME MULTIFIELD ADDRESS INSTANCE)
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>
```

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>
```

12.16.1.18 Getting the List of Facets for a Slot

This function returns a multifield listing the facet values for the specified slot (the slot can be inherited or explicitly defined for the class). A multifield of length zero is returned if an error occurs. Following is a table indicating what each field represents and its possible values:

Field	Meaning	Values	Explanation
1	Field Type	SGL/MLT	Single-field or multifield
2	Default Value	STC/DYN/NIL	Static, dynamic, or none
3	Inheritance	INH/NIL	Inheritable by other classes or not
4	Access	RW/R/INT	Read-write, read-only, or initialize-only
5	Storage	LCL/SHR	Local or shared
6	Pattern-Match	RCT/NIL	Reactive or non-reactive
7	Source	EXC/CMP	Exclusive or composite
8	Visibility	PUB/PRV	Public or private
9	Automatic Accessors	R/W/RW/NIL	Read, write, read-write, or none
10	Override-Message	<message-name>	Name of message sent for slot-overrides

See section 9.3.3 for more details on slot facets.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (slot x (access read-only)))
CLIPS> (defclass B (is-a A) (multislot y))
CLIPS> (slot-facets B x)
(SGL STC INH R SHR RCT EXC PRV R NIL)
CLIPS> (slot-facets B y)
```

```
(MLT STC INH RW LCL RCT EXC PRV RW put-y)
CLIPS>>
```

12.16.1.19 Getting the List of Source Classes for a Slot

This function groups the names of the classes which provide facets for a slot of a class into a multifield variable. In the case of an exclusive slot, this multifield will be of length one and contain the name of the contributing class. However, composite slots may have facets from many different classes (see section 9.3.3.6). A multifield of length zero is returned if an error occurs.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot x (access read-only)))
CLIPS>
(defclass B (is-a A)
  (slot x (source composite)
          (default 100)))
CLIPS> (defclass C (is-a B))
CLIPS> (slot-sources A x)
(A)
CLIPS> (slot-sources B x)
(A B)
CLIPS> (slot-sources C x)
(A B)
CLIPS>
```

12.16.1.20 Getting the Primitive Types for a Slot

This function groups the names of the primitive types allowed for a slot into a multifield variable. A multifield of length zero is returned if an error occurs.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (slot y (type INTEGER LEXEME)))
CLIPS> (slot-types A y)
(INTEGER SYMBOL STRING)
CLIPS>
```

12.16.1.21 Getting the Cardinality for a Slot

This function groups the minimum and maximum cardinality allowed for a multifield slot into a multifield variable. A maximum cardinality of infinity is indicated by the symbol **+oo** (the plus character followed by two lowercase o's—not zeroes). A multifield of length zero is returned for single field slots or if an error occurs.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (slot x)
  (multislot y (cardinality ?VARIABLE 5))
  (multislot z (cardinality 3 ?VARIABLE)))
CLIPS> (slot-cardinality A x)
()
CLIPS> (slot-cardinality A y)
(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>
```

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>
```

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.

12.16.1.26 Getting the Defaults Mode for Classes

This function returns the current defaults mode used when classes are defined (convenience or conservation).

Syntax

```
(get-class-defaults-mode)
```

12.16.1.27 Getting the Allowed Values for a Slot

This function groups the allowed classes for a slot (specified by the allowed-classes facet for the slot) into a multifield variable. If the allowed-classes facet was not specified for the slot, then the symbol FALSE is returned. A multifield of length zero is returned if an error occurs.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
```

```
CLIPS> (defclass B (is-a USER) (slot x))
CLIPS> (defclass C (is-a USER) (slot y (allowed-classes A B)))
CLIPS> (slot-allowed-classes B x)
FALSE
CLIPS> (slot-allowed-classes C y)
(A B)
CLIPS>
```

12.16.2 Message-handler Functions

12.16.2.1 Existence of Shadowed Handlers

This function returns the symbol TRUE if there is another message-handler available for execution, FALSE otherwise. If this function is called from an around handler and there are any shadowed handlers (see section 9.5.3), the return value is the symbol TRUE. If this function is called from a primary handler and there are any shadowed primary handlers, the return value is the symbol TRUE. In any other circumstance, the return value is the symbol FALSE.

Syntax

```
(next-handlerp)
```

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))
CLIPS>
(defmessage-handler A print-args ($?any)
  (println "A: " ?any)
  (if (next-handlerp) then
     (call-next-handler)))
```

```
CLIPS>
(defmessage-handler USER print-args ($?any)
  (println "USER: " ?any))
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>
```

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))
CLIPS>
(defmessage-handler A print-args ($?any)
  (println "A: " ?any)
  (if (next-handlerp) then
     (override-next-handler (rest$ ?any))))
CLIPS>
(defmessage-handler USER print-args ($?any)
  (println "USER: " ?any))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print-args 1 2 3 4)
A: (1 2 3 4)
USER: (2 3 4)
CLIPS>
```

12.16.3 Definstances Functions

12.16.3.1 Getting the List of Definstances

The function **get-definstances-list** returns a multifield value containing the names of all definstances 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 definstances are returned.

Syntax

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

Example

```
CLIPS> (clear)
CLIPS> (get-definstances-list)
CLIPS> (definstances foo)
CLIPS> (definstances bar)
CLIPS> (get-definstances-list)
(foo bar)
CLIPS>>
```

12.16.3.2 Determining the Module in which a Definstances is Defined

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

Syntax

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

12.16.4 Instance Manipulation Functions and Actions

12.16.4.1 Initializing an Instance

This function implements the init message-handler attached to the class USER (see section 9.4.5.1). This function evaluates and places slot expressions given by the class definition that were not specified by slot-overrides in the call to **make-instance** or **initialize-instance** (see section 9.6.1). This function should never be called directly unless an init message-handler is being defined such that the one attached to USER will never be called. However, such a definition is unusual and recommended only to advanced users. A user-defined class which does not inherit indirectly or directly from the class USER will require an init message-handler which calls this function in order for instances of the class to be created. If this function is called from an init message within the context of a **make-instance** or **initialize-instance** call and there are no errors in evaluating the class defaults, this function will return the address of the instance it is initializing. Otherwise, this function will return the symbol FALSE.

Syntax

```
(init-slots)
```

12.16.4.2 Deleting an Instance

This function deletes the specified instances by sending them a **delete** message. The argument can be one or more instance-names, instance-addresses, or symbols (an instance-name without enclosing brackets). The instance specified by the arguments must exist (except in the case of "*"). If "*" is specified for the instance, all instances will be sent the **delete** message (unless

there is an instance named "*"). This function returns the symbol TRUE if all instances were successfully deleted, otherwise it returns the symbol FALSE. Note, this function is exactly equivalent to sending the instance(s) the **delete** message directly and is provided only as an intuitive counterpart to the function **retract** for facts.

Syntax

```
(unmake-instance <instance-expression>+)
```

12.16.4.3 Deleting the Active Instance from a Handler

This function operates implicitly on the active instance (see section 9.4.1.1) for a message, and thus can only be called from within the body of a message-handler. This function directly deletes the active instance and is the one used to implement the delete handler attached to class USER (see section 9.4.5.2). This function returns the symbol TRUE if the instance was successfully deleted, otherwise the symbol FALSE.

Syntax

```
(delete-instance)
```

12.16.4.4 Determining the Class of an Object

This function returns a symbol which is the name of the class of its argument. It returns the symbol FALSE on errors. This function is equivalent to the **type** function.

Syntax

```
(class <object-expression>)
```

Example

```
CLIPS> (class 34)
INTEGER
CLIPS>
```

12.16.4.5 Determining the Name of an Instance

This function returns a symbol which is the name of its instance argument. It returns the symbol FALSE on errors. The evaluation of the argument must be an instance-name or instance-address of an existing instance.

Syntax

```
(instance-name <instance-expression>)
```

12.16.4.6 Determining the Address of an Instance

This function returns the address of its instance argument. It returns the symbol FALSE on errors. The evaluation of <instance expression> must be an instance-name or instance-address of an existing instance. If <module> or * is not specified, the function searches only in the current module. If * is specified, the current module and imported modules are recursively searched. If <module> is specified, only that module is searched. The :: syntax cannot be used with the instance-name if <module> or * is specified.

Syntax

```
(instance-address [<module> | *] <instance-expression>)
```

12.16.4.7 Converting a Symbol to an Instance-Name

This function returns an instance-name which is equivalent to its symbol argument. It returns the symbol FALSE on errors.

Syntax

```
(symbol-to-instance-name <symbol-expression>)
```

Example

```
CLIPS> (symbol-to-instance-name (sym-cat abc def))
[abcdef]
CLIPS>
```

12.16.4.8 Converting an Instance-Name to a Symbol

This function returns a symbol which is equivalent to its instance-name argument. It returns the symbol FALSE on errors.

Syntax

```
(instance-name-to-symbol <instance-name-expression>)
```

Example

```
CLIPS> (instance-name-to-symbol [a])
a
CLIPS>
```

12.16.4.9 Predicate Functions

12.16.4.9.1 Testing for an Instance

This function returns the symbol TRUE if the evaluation of its argument is an instance-address or an instance-name. Otherwise, it returns the symbol FALSE.

Syntax

```
(instancep <expression>)
```

12.16.4.9.2 Testing for an Instance-Address

This function returns the symbol TRUE if the evaluation of its argument is an instance-address. Otherwise, it returns the symbol FALSE.

Syntax

```
(instance-addressp <expression>)
```

12.16.4.9.3 Testing for an Instance-Name

This function returns the symbol TRUE if the evaluation of its argument is an instance-name. Otherwise, it returns the symbol FALSE.

Syntax

```
(instance-namep <expression>)
```

12.16.4.9.4 Testing for the Existence an Instance

This function returns the symbol TRUE if the specified instance exists. Otherwise, it returns the symbol FALSE. If the argument is an instance-name, the function determines if an instance of the specified name exists. If the argument is an instance-address, the function determines if the specified address is still valid.

Syntax

```
(instance-existp <instance-expression>)
```

12.16.4.10 Reading a Slot Value

This function returns the value of the specified slot of the active instance (see section 9.4.1.1). If the slot does not exist, the slot does not have a value or this function is called from outside a message-handler, this function will return the symbol FALSE and an error will be generated. This function differs from the ?self:<slot-name> syntax in that the slot is not looked up until the function is actually called. Thus it is possible to access different slots every time the function is executed (see section 9.4.2 for more details). This function bypasses message-passing.

Syntax

```
(dynamic-get <slot-name-expression>)
```

12.16.4.11 Setting a Slot Value

This function sets the value of the specified slot of the active instance (see section 9.4.1.1). If the slot does not exist, there is an error in evaluating the arguments to be placed or this function is called from outside a message-handler, this function will return the symbol FALSE and an error will be generated. Otherwise, the new slot value is returned. This function differs from the (bind ?self:<slot-name> <value>*) syntax in that the slot is not looked up until the function is actually called. Thus it is possible to access different slots every time the function is executed (see section 9.4.2 for more details). This function bypasses message-passing.

Syntax

```
(dynamic-put <slot-name-expression> <expression>*)
```

12.16.4.12 Multifield Slot Functions

The following functions allow convenient manipulation of multifield slots. There are three types of functions: replacing a range of fields with one or more new values, inserting one or more new values at an arbitrary point, and deleting a range of fields. For each type, there are two forms of functions: an external interface which sets the new value for the multifield slot with a put- message (see section 9.3.3.9), and an internal interface that can only be called from message-handlers which sets the slot for the active instance (see section 9.4.1.1) directly. Both forms read the original value of the slot directly without the use of a get- message. All of these functions return the new slot value on success and the symbol FALSE on errors.

12.16.4.12.1 Replacing Fields

Allows the replacement of a range of fields in a multifield slot value with one or more new values. The range indices must be from 1..n, where n is the number of fields in the multifield slot's original value and n > 0.

External Interface Syntax

```
(slot-replace$ <instance-expression> <mv-slot-name>
  <range-begin> <range-end> <expression>+)
```

Internal Interface Syntax

```
(slot-direct-replace$ <mv-slot-name>
  <range-begin> <range-end> <expression>+)
```

Example

```
CLIPS>
(defclass A (is-a USER)
   (multislot mfs (default a b c d e)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (slot-replace$ a mfs 2 4 2 3 4)
(a 2 3 4 e)
CLIPS>
```

12.16.4.12.2 Inserting Fields

Allows the insertion of one or more new values in a multifield slot value before a specified field index. The index must greater than or equal to 1. A value of 1 inserts the new value(s) at the beginning of the slot's value. Any value greater than the length of the slot's value appends the new values to the end of the slot's value.

External Interface Syntax

```
(slot-insert$ <instance-expression> <mv-slot-name>
  <index> <expression>+)
```

Internal Interface Syntax

```
(slot-direct-insert$ <mv-slot-name> <index> <expression>+)
```

Example

```
CLIPS> (initialize-instance a)
[a]
CLIPS> (slot-insert$ a mfs 2 4 2 3 4)
(a 4 2 3 4 b c d e)
CLIPS>
```

12.16.4.12.3 Deleting Fields

Allows the deletion of a range of fields in a multifield slot value. The range indices must be from 1..n, where n is the number of fields in the multifield slot's original value and n > 0.

External Interface Syntax

```
(slot-delete$ <instance-expression> <mv-slot-name>
  <range-begin> <range-end>)
```

Internal Interface Syntax

```
(slot-direct-delete$ <mv-slot-name> <range-begin> <range-end>)
```

Example

```
CLIPS> (initialize-instance a)
[a]
CLIPS> (slot-delete$ a mfs 2 4)
(a e)
CLIPS>
```

12.17 Defmodule Functions

The following functions provide ancillary capabilities for the defmodule construct.

12.17.1 Getting the List of Defmodules

The function **get-defmodule-list** returns a multifield value containing the names of all defmodules that are currently defined.

Syntax

```
(get-defmodule-list)
```

Example

```
CLIPS> (clear)
CLIPS> (get-defmodule-list)
(MAIN)
CLIPS> (defmodule A)
CLIPS> (defmodule B)
CLIPS> (get-defmodule-list)
(MAIN A B)
CLIPS>
```

12.17.2 Setting the Current Module

This function sets the current module. It returns the name of the previous current module. If an invalid module name is given, then the current module is not changed and the name of the current module is returned.

Syntax

```
(set-current-module <defmodule-name>)
```

12.17.3 Getting the Current Module

This function returns the name of the current module.

Syntax

```
(get-current-module)
```

12.18 Sequence Expansion

By default, there is no distinction between single-field and multifield variable references within function calls (as opposed to declaring variables for function parameters or variables used for pattern-matching). For example:

```
CLIPS> (clear)
CLIPS>
(defrule expansion
  (foo $?b)
  =>
  (println ?b)
  (println $?b))
CLIPS> (assert (foo a b c))
<Fact-1>
CLIPS> (run)
(a b c)
(a b c)
CLIPS>
```

Note that both printout statements in the rule produce identical output when the rule executes. The use of ?b and $?b within the function call behave identically.

Multifield variable references within function calls, however, can optionally be expanded into multiple single field arguments. The $ acts as a “sequence expansion” operator and has special meaning when applied to a global or local variable reference within the argument list of a function call. The $ means to take the fields of the multifield value referenced by the variable and treat them as separate arguments to the function as opposed to passing a single multifield value argument.

For example, using sequence expansion with the *expansion* rule would give the following output:

```
CLIPS> (clear)
CLIPS> (set-sequence-operator-recognition TRUE)
TRUE
CLIPS>
(defrule expansion
```

```
   (foo $?b)
   =>
   (println ?b)
   (println $?b))
CLIPS> (assert (foo a b c))
<Fact-1>
CLIPS> (run)
(a b c)
abc
CLIPS> (set-sequence-operator-recognition FALSE)
TRUE
CLIPS>
```

Using sequence expansion, the two printout statements on the RHS of the expansion rule are equivalent to:

```
(printout t (create$ a b c) crlf)
(printout t a b c crlf)
```

The $ operator also works with global variables. For example:

```
CLIPS> (clear)
CLIPS> (set-sequence-operator-recognition TRUE)
FALSE
CLIPS> (defglobal ?*x* = (create$ 3 4 5))
CLIPS> (+ ?*x*)
[ARGACCES4] Function + expected at least 2 argument(s)
CLIPS> (+ $?*x*)
12
CLIPS> (set-sequence-operator-recognition FALSE)
TRUE
CLIPS>
```

The sequence expansion operator is particularly useful for generic function methods. Consider the ease now of defining a general addition function for strings.

```
(defmethod + (($?any STRING))
   (str-cat $?any))
```

By default, sequence expansion is disabled. This allows previously existing CLIPS programs to work correctly with version 6.0 of CLIPS. The behavior can be enabled using the **set-sequence-operator-recognition** function described in section 12.18.3. Old CLIPS code should be changed so that it works properly with sequence expansion enabled.

12.18.1 Sequence Expansion and Rules

Sequence expansion is allowed on the LHS of rules, but only within function calls. If a variable is specified in a pattern as a single or multifield variable, then all other references to that variable that are not within function calls must also be the same. For example, the following rule is not allowed

```
(defrule bad-rule-1
  (pattern $?x ?x $?x)
  =>)
```

The following rules illustrate appropriate use of sequence expansion on the LHS of rules.

```
(defrule good-rule-1
  (pattern $?x&:(> (length$ ?x) 1))
  (another-pattern $?y&:(> (length$ ?y) 1))
  (test (> (+ $?x) (+ $?y)))
  =>)
```

The first and second patterns use the length$ function to determine that the multifields bound to ?x and ?y are greater than 1. Sequence expansion is not used to pass ?x and ?y to the length$ function since the length$ function expects a single argument of type multifield. The test CE calls the + function to determine the sum of the values bound to ?x and ?y. Sequence expansion is used for these function calls since the + function expects two or more arguments with numeric data values.

Sequence expansion has no affect within an **assert**, **modify**, or **duplicate**; however, it can be used with other functions on the RHS of a rule.

12.18.2 Multifield Expansion Function

The $ operator is merely a shorthand notation for the **expand$** function call. For example, the function calls

```
(println $?b)
```

and

```
(println (expand$ ?b))
```

are identical.

Syntax

```
(expand$ <multifield-expression>)
```

The **expand$** function is valid only within the argument list of a function call. The **expand$** function (and hence sequence expansion) cannot be used as an argument to the following functions: **expand$**, **return**, **progn**, **while**, **if**, **progn$**, **foreach**, **switch**, **loop-for-count**, **assert**, **modify**, **duplicate** and **object-pattern-match-delay**.

12.18.3 Setting The Sequence Operator Recognition Behavior

This function sets the sequence operator recognition behavior. When this behavior is disabled (FALSE by default), multifield variables found in function calls are treated as a single argument. When this behaviour is enabled, multifield variables are expanded and passed as separate arguments in the function call. This behavior should be set *before* an expression references a multifield variable is encountered (i.e. changing the behavior does not retroactively change the behavior for previously encountered expressions). The return value for this function is the old value for the behavior.

Syntax

```
(set-sequence-operator-recognition <boolean-expression>)
```

12.18.4 Getting The Sequence Operator Recognition Behavior

This function returns the current value of the sequence operator recognition behavior (TRUE or FALSE).

Syntax

```
(get-sequence-operator-recognition)
```

12.18.5 Sequence Operator Caveat

CLIPS normally tries to detect as many constraint errors as possible for a function call at parse time, such as bad number of arguments or types. However, if the sequence expansion operator is used in the function call, all such checking is delayed until run-time (because the number and types of arguments can change for each execution of the call.) For example:

```
CLIPS> (clear)
CLIPS> (set-sequence-operator-recognition TRUE)
FALSE
CLIPS> (deffunction foo (?a ?b))
CLIPS> (deffunction bar ($?a) (foo ?a))
[ARGACCES4] Function foo expected exactly 2 argument(s)

ERROR:
(deffunction MAIN::bar
   ($?a)
   (foo ?a)
CLIPS> (deffunction bar ($?a) (foo $?a))
CLIPS> (bar 1)
[ARGACCES4] Function foo expected exactly 2 argument(s)
[PRCCODE4] Execution halted during the actions of deffunction bar.
FALSE
CLIPS> (bar 1 2)
FALSE
CLIPS> (set-sequence-operator-recognition FALSE)
```

```
TRUE
CLIPS>
```

Section 13:
Commands

This section describes commands primarily intended for use from the top-level command prompt. These commands may also be used from constructs and other places where functions can be used.

13.1 Environment Commands

The following commands control the CLIPS environment.

13.1.1 Loading Constructs From A File

Loads the constructs stored in the file specified by <file-name> into the environment. If the compilations item is being watched (see section 13.2.3), then an informational message (including the type and name of the construct) will be displayed for each construct loaded. If the compilations item is not being watched, then a character is printed for each construct loaded ("*" for defrule, "$" for deffacts, "%" for deftemplate, ":" for defglobal, "!" for deffunction, "^" for defgeneric, "&" for defmethod, "#" for defclass, "~" for defmessage-handler, "@" for definstances, and "+" for defmodule). This function returns TRUE if the file was successfully loaded, otherwise FALSE is returned.

Syntax

```
(load <file-name>)
```

13.1.2 Loading Constructs From A File without Progress Information

Loads the constructs stored in the file specified by <file-name> into the environment, however, unlike the load command informational messsages are not printed to show the progress of loading the file. Error messages are still printed if errors are encountered while loading the file. This function returns TRUE if the file was successfully loaded, otherwise FALSE is returned.

Syntax

```
(load* <file-name>)
```

13.1.3 Saving All Constructs To A File

Saves all of the constructs (defrules, deffacts, deftemplates, etc.) in the current environment into the file specified by <file-name>. Note that deffunctions and generic functions are saved twice to the file. Because it is possible to create circular references among deffunctions and generic functions by redefining them, a forward declaration (containing no actions) of each function is saved first to the file, and then the actual declaration (containing the actions) is saved. This function returns TRUE if the file was successfully saved, otherwise FALSE is returned. This function uses the pretty-print forms of the constructs. If pretty-printing has been disabled by the **conserve-mem** command, then the **save** command will have no output.

Syntax

```
(save <file-name>)
```

13.1.4 Loading a Binary Image

Loads the constructs stored in the binary file specified by <file-name> into the environment. The specified file must have been created by **bsave**. Loading a binary image is quicker than using the **load** command to load a ASCII text file. A **bload** clears all constructs from the current CLIPS environment (as well as all facts and instances). **Bload** can be called at any time unless some constructs that **bload** will affect are in use (e.g. a deffunction is currently executing). The only constructive/destructive operation that can occur after a **bload** is the **clear** command or the **bload** command (which clears the current binary image). This means that constructs cannot be loaded or deleted while a **bload** is in effect. In order to add constructs to a binary image, the original ASCII text file must be reloaded, the new constructs added, and then another **bsave** must be performed. This function returns TRUE if the file was successfully bloaded, otherwise FALSE is returned.

Binary images can be loaded into different compile-time configurations of CLIPS, as long as the same version of CLIPS is used and all the functions and constructs needed by the binary image are supported. In addition, binary images should theoretically work across different hardware platforms if internal data representations are equivalent (e.g. same integer size, same byte order, same floating-point format, etc). However, it is NOT recommended that this be attempted.

Syntax

```
(bload <file-name>)
```

13.1.5 Saving a Binary Image

Saves all of the constructs in the current environment into the file specified by <file-name>. The save file is written using a binary format which results in faster load time. The save file must be loaded via the **bload** command. A **bsave** may be performed at any time (except when a **bload** is

in effect). The pretty print representation of a construct is not saved with a binary image (thus, commands like **ppdefrule** will show no output for any of the rules associated with the binary image). In addition, constraint information associated with constructs is not saved to the binary image unless dynamic constraint checking is enabled (using the **set-dynamic-constraint-checking** command). This function returns TRUE if the file was successfully bsaved, otherwise FALSE is returned.

Syntax

```
(bsave <file-name>)
```

13.1.6 Clearing CLIPS

Clears CLIPS. Removes all constructs and all associated data structures (such as facts and instances) from the CLIPS environment. A clear may be performed safely at any time, however, certain constructs will not allow themselves to be deleted while they are in use. For example, while deffacts are being reset (by the **reset** command), it is not possible to remove them using the **clear** command. Note that the **clear** command does not effect many environment characteristics (such as the current conflict resolution strategy). This function has no return value.

Syntax

```
(clear)
```

13.1.7 Exiting CLIPS

Quits CLIPS. This function has no return value.

Syntax

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

The optional <integer-expression> argument allows the exit status code to be specified which is eventually passed to the C exit function.

13.1.8 Resetting CLIPS

Resets CLIPS. Removes all activations from the agenda, all facts from the fact-list and all instances of user-defined classes, then assigns global variables their initial values, asserts all facts listed in deffacts statements into the fact-list, creates all instances listed in definstances statements, sets the current module to the MAIN module and automatically focuses on the same module. Incremental reset is supported for rules. This means that rules can be activated from

facts that were asserted before the rule was defined without performing a reset. A reset can be performed while rules are executing. Note that the **reset** command does not effect many environment characteristics (such as the current conflict resolution strategy). This function has no return value.

Syntax

```
(reset)
```

13.1.9 Executing Commands From a File

Allows "batch" processing of CLIPS interactive commands by replacing standard input with the contents of a file. Any command or function can be used in a batch file, as well as construct definitions and responses to **read** or **readline** function calls. The **load** command should be used in batch files rather than defining constructs directly. The **load** command expects only constructs and hence moves to the next construct when an error occurs. The **batch** command, however, moves on until it finds the next construct *or* command (and in the case of a construct this is likely to generate more errors as the remaining commands and functions in the construct are parsed). This function returns TRUE if the batch file was successfully executed, otherwise FALSE is returned. Note that the **batch** command operates by replacing standard input rather than by immediately executing the commands found in the batch file. In effect, if you execute a batch command from the RHS of a rule, the commands in that batch file will not be processed until control is returned to the top-level prompt.

Syntax

```
(batch <file-name>)
```

13.1.10 Executing Commands From a File Without Replacing Standard Input

Evaluates the series of commands stored in the file specified by <file-name>. Unlike the **batch** command, **batch*** evaluates all of the commands in the specified file before returning. The **batch*** command does not replace standard input and thus a **batch*** file cannot be used to provide input to functions such as **read** and **readline**. In addition, commands stored in the **batch*** file and the return value of these commands are not echoed to standard output.

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

Syntax

```
(batch* <file-name>)
```

13.1.11 Determining CLIPS Compilation Options

Generates a textual description of the settings of the CLIPS compiler flags. This function has no return value.

Syntax

```
(options)
```

13.1.12 Calling the Operating System

The **system** function allows a call to the operating system. If no arguments are specified, the function returns 0 if a command processer is unavailable; otherwise, it returns a non-zero value. If one or more string/symbol arguments are specified, the arguments are concatenated into a single command string and this string is then passed to the command processor. In this case, the function returns an integer value indicating the completion status of the command (which can vary depending upon your operating system and compiler). If any invalid arguments are specified, this function returns the symbol FALSE.

Syntax

```
(system <lexeme-expression>*)
```

Example

```
(defrule print-directory
   (print-directory  ?directory)
   =>
   (system "dir " ?directory));      Note space => "dir<space>"
```

Note that any spaces needed for a proper parsing of the **system** command must be added by the user in the call to system. Also note that the system command is not guaranteed to execute (e.g., the operating system may not have enough memory to spawn a new process).

❖ Portability Note

The **system** function uses the ANSI C function **system** as a base. The return value of this ANSI library function is implementation dependent and may change for different operating systems and compilers.

13.1.13 Setting the Dynamic Constraint Checking Behavior

This function sets dynamic constraint checking behavior. When this behavior is disabled (FALSE by default), newly created data objects (such as deftemplate facts and instances) do not have their slot values checked for constraint violations. When this behavior is enabled (TRUE),

the slot values are checked for constraint violations. The return value for this function is the old value for the behavior. Constraint information is not saved when using the **bload** and **constructs-to-c** command if dynamic constraint checking is disabled.

Syntax

```
(set-dynamic-constraint-checking <boolean-expression>)
```

13.1.14 Getting the Dynamic Constraint Checking Behavior

This function returns the current value of the dynamic constraint checking behavior (TRUE or FALSE).

Syntax

```
(get-dynamic-constraint-checking)
```

13.1.15 Finding Symbols

This command displays all symbols currently defined in CLIPS which contain a specified substring. This command has no return value.

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 **stdout**, **werror**, and **wwarning** 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.

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 |
                 messages |
                 deffunctions <deffunction-name>* |
                 globals <global-name>* |
                 rules <rule-name>* |
                 activations <rule-name>* |
                 facts <deftemplate-name>* |
                 instances <class-name>* |
                 slots <class-name>* |
                 message-handlers <handler-spec-1>*
                                  [<handler-spec-2>]) |
                 generic-functions <generic-name>* |
                 methods <method-spec-1>* [<method-spec-2>]

<handler-spec-1> ::= <class-name>
                                <handler-name> <handler-type>
<handler-spec-2> ::= <class-name>
                              [<handler-name> [<handler-type>]]

<method-spec-1> ::= <generic-name> <method-index>
<method-spec-2> ::= <generic-name> [<method-index>]
```

If **compilations** are watched, the progress of construct definitions will be displayed.

If **facts** are watched, all fact assertions and retractions will be displayed. Optionally, facts associated with individual deftemplates can be watched by specifying one or more deftemplate names.

If **rules** are watched, all rule firings will be displayed. If **activations** are watched, all rule activations and deactivations will be displayed. Optionally, rule firings and activations associated with individual defrules can be watched by specifying one or more defrule names. If **statistics** are watched, timing information along with other information (average number of facts, average number of activations, etc.) will be displayed after a run. Note that the number of rules fired and timing information is not printed unless this item is being watch. If **focus** is watched, then changes to the current focus will be displayed.

If **globals** are watched, variable assignments to globals variables will be displayed. Optionally, variable assignments associated with individual defglobals can be watched by specifying one or more defglobal names. If **deffunctions** are watched, the start and finish of deffunctions will be displayed. Optionally, the start and end display associated with individual deffunctions can be watched by specifying one or more deffunction names.

If **generic-functions** are watched, the start and finish of generic functions will be displayed. Optionally, the start and end display associated with individual defgenerics can be watched by specifying one or more defgeneric names. If **methods** are watched, the start and finish of individual methods within a generic function will be displayed. Optionally, individual methods can be watched by specifying one or more methods using a defgeneric name and a method index. When the method index is not specified, then all methods of the specified defgeneric will be watched.

If **instances** are watched, creation and deletion of instances will be displayed. If **slots** are watched, changes to any instance slot values will be displayed. Optionally, instances and slots associated with individual concrete defclasses can be watched by specifying one or more concrete defclass names. If **message-handlers** are watched, the start and finish of individual message-handlers within a message will be displayed. Optionally, individual message-handlers can be watched by specifying one or more message-handlers using a defclass name, a message-handler name, and a message-handler type. When the message-handler name and message-handler type are not specified, then all message-handlers for the specified class will be watched. When the message-handler type is not specified, then all message-handlers for the specified class with the specified message-handler name will be watched. If **messages** are watched, the start and finish of messages will be displayed.

For the watch items that allow individual constructs to be watched, if no constructs are specified, then all constructs of that type will be watched. If all constructs associated with a watch item are being watched, then newly defined constructs of the same type will also be watched. A construct retains its old watch state if it is redefined. If **all** is watched, then all other watch items will be watched. By default, no items are watched. The watch function has no return value.

Example

```
CLIPS> (watch rules)
CLIPS>
```

13.2.4 Disabling Watch Items

This function disables the effect of the **watch** command.

Syntax

```
(unwatch <watch-item>)
```

This command is identical to the watch command with the exception that it disables watch items rather than enabling them. This function has no return value.

Example

```
CLIPS> (unwatch all)
CLIPS>
```

13.2.5 Viewing the Current State of Watch Items

This command displays the current state of watch items.

Syntax

```
(list-watch-items [<watch-item>])
```

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 = off
statistics = off
globals = off
focus = off
CLIPS> (list-watch-items facts)
facts = off
MAIN:
CLIPS>
```

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 module are displayed. If <module-name> is the symbol *, then the names of all deftemplates in all modules are displayed.

13.3.3 Deleting a Deftemplate

This function deletes a previously defined deftemplate.

Syntax

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

If the deftemplate is in use (for example by a fact or a rule), then the deletion will fail. Otherwise, no further uses of the deleted deftemplate are permitted (unless redefined). If the symbol * is used for <deftemplate-name>, then all deftemplates will be deleted (unless there is a deftemplate named *). This function has no return value.

13.4 Fact Commands

The following commands display information about facts.

13.4.1 Displaying the Fact-List

Displays facts stored in the fact-list.

Syntax

```
(facts [<module-name>]
       [<start-integer-expression>
        [<end-integer-expression>
         [<max-integer-expression>]]])
```

If <module-name> is not specified, then only facts visible to the current module will be displayed. If <module-name> is specified, then only facts visible to the specified module are displayed. If the symbol * is used for <module-name>, then facts from any module may be displayed. If the start argument is specified, only facts with fact-indices greater than or equal to this argument are displayed. If the end argument is specified, only facts with fact-indices less than or equal to this argument are displayed. If the max argument is specified, then no facts will be displayed beyond the specified maximum number of facts to be displayed. This function has no return value.

13.4.2 Loading Facts From a File

This function will assert a file of information as facts into the CLIPS fact-list. It can read files created with save-facts or any ASCII text file. Each fact should begin with a left parenthesis and end with a right parenthesis. Facts may span across lines and can be written in either ordered or deftemplate format. This function returns TRUE if the fact file was successfully loaded, otherwise FALSE is returned.

Syntax

```
(load-facts <file-name>)
```

13.4.3 Saving The Fact-List To A File

This function saves all of the facts in the current fact-list into the file specified by <file-name>. External-address and fact-address fields are saved as strings. Instance-address fields are converted to instance-names. Optionally, the scope of facts to be saved can be specified. If <save-scope> is the symbol **visible**, then all facts visible to the current module are saved. If <save-scope> is the symbol **local**, then only those facts with deftemplates defined in the current module are saved. If <save-scope> is not specified, it defaults to **local**. If <save-scope> is specified, then one or more deftemplate names may also be specified. In this event, only those facts with associated with a corresponding deftemplate in the specified list will be saved. This function returns TRUE if the fact file was successfully saved, otherwise FALSE is returned.

Syntax

```
(save-facts <file-name> [<save-scope> <deftemplate-names>*])

<save-scope> ::= visible | local
```

13.4.4 Setting the Duplication Behavior of Facts

This function sets fact duplication behavior. When this behavior is disabled (FALSE by default), asserting a duplicate of a fact already in the fact-list produces no effect. When enabled (TRUE), the duplicate fact is asserted with a new fact-index. The return value for this function is the old value for the behavior.

Syntax

```
(set-fact-duplication <boolean-expression>)
```

Example

```
CLIPS> (clear)
CLIPS> (get-fact-duplication)
FALSE
CLIPS> (watch facts)
CLIPS> (assert (a))
==> f-1     (a)
<Fact-1>
CLIPS> (assert (a))
FALSE
CLIPS> (set-fact-duplication TRUE)
FALSE
CLIPS> (assert (a))
==> f-2     (a)
<Fact-2>
CLIPS> (unwatch facts)
CLIPS> (set-fact-duplication FALSE)
TRUE
CLIPS>
```

13.4.5 Getting the Duplication Behavior of Facts

This function returns the current value of the fact duplication behavior (TRUE or FALSE).

Syntax

```
(get-fact-duplication)
```

13.4.6 Displaying a Single Fact

Displays a single fact, placing each slot and its value on a separate line. Optionally the logical name to which output is sent can be specified and slots containing their default values can be

excluded from the output. If <logical-name> is **t** or unspecified, then output is sent to the logical name **stdout**, otherwise it is sent to the specified logical name. If <ignore-defaults-flag> is FALSE or unspecified, then all of the fact's slots are displayed, otherwise slots with static defaults are only displayed if their current slot value differs from their initial default value.

Syntax

```
(ppfact <fact-specifier> [<logical-name> [<ignore-defaults-flag>]])
```

Example

```
CLIPS> (clear)
CLIPS>
(deftemplate foo
   (slot x (default 3))
   (slot y)
   (multislot z (default a b)))
CLIPS> (assert (foo))
<Fact-1>
CLIPS> (ppfact 1 t)
(foo
   (x 3)
   (y nil)
   (z a b))
CLIPS> (ppfact 1 t TRUE)
(foo)
CLIPS> (modify 1 (y 2) (z c))
<Fact-1>
CLIPS> (ppfact 1 t TRUE)
(foo
   (y 2)
   (z c))
CLIPS>
```

13.5 Deffacts Commands

The following commands manipulate deffacts.

13.5.1 Displaying the Text of a Deffacts

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

Syntax

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

13.5.2 Displaying the List of Deffacts

Displays the names of all deffacts stored in the CLIPS environment.

Syntax

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

If <module-name> is unspecified, then the names of all deffacts in the current module are displayed. If <module-name> is specified, then the names of all deffacts in the specified module are displayed. If <module-name> is the symbol *, then the names of all deffacts in all modules are displayed. This function has no return value.

13.5.3 Deleting a Deffacts

This function deletes a previously defined deffacts.

Syntax

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

All facts listed in the deleted deffacts construct will no longer be asserted as part of a reset. If the symbol * is used for <deffacts-name>, then all deffacts will be deleted (unless there exists a deffacts named *). The **undeffacts** command can be used to remove deffacts at any time. Exceptions: When deffacts are being reset as part of the **reset** command, they cannot be removed. This function has no return value.

13.6 Defrule Commands

The following commands manipulate defrules.

13.6.1 Displaying the Text of a Rule

Displays the text of a given rule.

Syntax

```
(ppdefrule <rule-name>)
```

The **pprule** command can also be used for this purpose. This function has no return value.

13.6.2 Displaying the List of Rules

Displays the names of all rules stored in the CLIPS environment.

Syntax

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

If <module-name> is unspecified, then the names of all defrules in the current module are displayed. If <module-name> is specified, then the names of all defrules in the specified module are displayed. If <module-name> is the symbol *, then the names of all defrules in all modules are displayed. This function has no return value.

13.6.3 Deleting a Defrule

This function deletes a previously defined defrule.

Syntax

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

If the defrule is in use (for example if it is firing), then the deletion will fail. If the symbol * is used for <defrule-name>, then all defrule will be deleted (unless there is a defrule named *). This function has no return value.

13.6.4 Displaying Matches for a Rule

For a specified rule, displays the list of the facts or instances which match each pattern in the rule's LHS, the partial matches for the rule, and the activations for the rule. When listed as a partial match, the *not*, *exists*, and *forall* CEs are shown as an asterisk. Other CEs contained within these CEs are not displayed as part of the information shown for a partial match. This function returns FALSE if the specified rule does not exist or the function is passed invalid arguments, otherwise a multifield value is returned containing three values: the combined sum of the matches for each pattern, the combined sum of partial matches, and the number of activations.

Syntax

```
(matches <rule-name> [<verbosity>])
```

where <verbosity> is either verbose, succinct, or terse. If <verbosity> is not specified or <verbosity> is verbose, then output will include details for each match, partial match, and activation. If <verbosity> is succinct, then output will just include the total number of matches, partial matches, and activations. If <verbosity> is terse, no output will be displayed.

Example

The rule *matches-example-1* has three patterns and none are added by CLIPS. Fact f-1 matches the first pattern, facts f-2 and f-3 match the the second pattern, and fact f-4 matches the third pattern. Issuing the run command will remove all of the rule's activations from the agenda.

```
CLIPS> (clear)
CLIPS>
(defrule matches-example-1
   (a ?)
   (b ?)
   (c ?)
   =>)
CLIPS> (reset)
CLIPS> (assert (a 1) (b 1) (b 2) (c 1))
<Fact-4>
CLIPS> (facts)
f-1     (a 1)
f-2     (b 1)
f-3     (b 2)
f-4     (c 1)
For a total of 4 facts.
CLIPS> (run)
CLIPS>
```

The rule *matches-example-2* has three patterns. There are no matches for the first pattern (since there are no *d* facts), facts f-2 and f-3 match the third pattern, and fact f-4 matches the forth pattern.

```
CLIPS>
(defrule matches-example-2
   (not (d ?))
   (exists (b ?x)
           (c ?x))
   =>)
CLIPS>
```

Listing the matches for the rule *matches-example-1* displays the matches for the patterns indicated previously. There are two partial matches which satisfy the first two patterns and two partial matches which satisfy all three patterns. Since all of the rule's activations were allowed to fire there are none listed.

```
CLIPS> (matches matches-example-1)
Matches for Pattern 1
f-1
Matches for Pattern 2
f-2
f-3
Matches for Pattern 3
f-4
Partial matches for CEs 1 - 2
f-1,f-3
f-1,f-2
Partial matches for CEs 1 - 3
f-1,f-2,f-4
f-1,f-3,f-4
Activations
 None
(4 4 0)
CLIPS>
```

Listing the matches for the rule *matches-example-2* displays the matches for the patterns indicated previously. There is one partial match which satisfies the first two CEs (the *not* pattern and the *exists* CE). The * indicates an existential match that is not associated with specific facts/instances (e.g. the *not* CE is satisfied because there are no *d* facts matching the pattern so * is used to indicate a match as there's no specific fact matching that pattern). Since none of the rule's activations were allowed to fire they are listed. The list of activations will always be a subset of the partial matches for all of the rule's CEs.

```
CLIPS> (matches matches-example-2)
Matches for Pattern 1
 None
Matches for Pattern 2
f-2
f-3
Matches for Pattern 3
f-4
Partial matches for CEs 1 - 2
*,f-2
*,f-3
Partial matches for CEs 1 - 3
*,f-2,f-4
Partial matches for CEs 1 (P1) , 2 (P2 - P3)
*,*
Activations
*,*
(3 4 1)
CLIPS>
```

If you just want a summary of the partial matches, specify succinct or terse as the second argument to the function.

```
CLIPS> (matches matches-example-2 succinct)
Pattern 1: 0
Pattern 2: 2
Pattern 3: 1
CEs 1 - 2: 2
CEs 1 - 3: 1
CEs 1 (P1) , 2 (P2 - P3): 1
Activations: 1
(3 4 1)
CLIPS> (matches matches-example-2 terse)
(3 4 1)
CLIPS>
```

13.6.5 Setting a Breakpoint for a Rule

Sets a breakpoint for a given rule.

Syntax

```
(set-break <rule-name>)
```

If a breakpoint is set for a given rule, execution will stop prior to executing that rule. At least one rule must fire before a breakpoint will stop execution. This function has no return value.

13.6.6 Removing a Breakpoint for a Rule

Removes a breakpoint for a given rule.

Syntax

```
(remove-break [<defrule-name>])
```

If no argument is given, then all breakpoints are removed. This function has no return value.

13.6.7 Displaying Rule Breakpoints

This command displays all the rules which have breakpoints set. This function has no return value.

Syntax

```
(show-breaks [<module-name>])
```

If <module-name> is unspecified, then the names of all rules having breakpoints in the current module are displayed. If <module-name> is specified, then the names of all rules having breakpoints in the specified module are displayed. If <module-name> is the symbol *, then the names of all rules having breakpoints in all modules are displayed.

13.6.8 Refreshing a Rule

Places all current activations of a given rule on the agenda. This function has no return value.

Syntax

```
(refresh <rule-name>)
```

13.6.9 Determining the Logical Dependencies of a Pattern Entity

The **dependencies** function lists the partial matches from which a pattern entity receives logical support. This function has no return value.

Syntax

```
(dependencies <fact-or-instance-specifier>)
```

The term <fact-or-instance-specifier> includes variables bound on the LHS to fact-addresses or instance-addresses as described in section 5.4.1.8, the fact-index of the desired fact (e.g. 3 for the fact labeled f-3), or the instance-name (e.g. [object]).

Example

```
(defrule list-dependencies
   ?f <- (factoid $?)
   =>
   (dependencies ?f))
```

13.6.10 Determining the Logical Dependents of a Pattern Entity

The **dependents** function lists all pattern entities which receive logical support from a pattern entity. This function has no return value.

Syntax

```
(dependents <fact-or-instance-specifier>)
```

The term <fact-or-instance-specifier> includes variables bound on the LHS to fact-addresses or instance-addresses as described in section 5.4.1.8, the fact-index of the desired fact (e.g. 3 for the fact labeled f-3), or the instance-name (e.g. [object]).

Example

```
(defrule list-dependents
   ?f <- (factoid $?)
   =>
   (dependents ?f))
```

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 13.2.3). 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.

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

```
(focus <module-name>+)
```

13.7.4 Stopping Rule Execution

The **halt** function may be used on the RHS of a rule to prevent further rule firing. It is called without arguments. After **halt** is called, control is returned from the **run** command. The agenda is left intact, and execution may be continued with a **run** command. This function has no return value.

Syntax

```
(halt)
```

13.7.5 Setting The Current Conflict Resolution Strategy

This function sets the current conflict resolution strategy. The default strategy is depth.

Syntax

```
(set-strategy <strategy>)
```

where <strategy> is either depth, breadth, simplicity, complexity, lex, mea, or random. The old conflict resolution strategy is returned. The agenda will be reordered to reflect the new conflict resolution strategy.

13.7.6 Getting The Current Conflict Resolution Strategy

This function returns the current conflict resolution strategy (depth, breadth, simplicity, complexity, lex, mea, or random).

Syntax

```
(get-strategy)
```

13.7.7 Listing the Module Names on the Focus Stack

The command **list-focus-stack** list all module names on the focus stack. The first name listed is the current focus.

Syntax

```
(list-focus-stack)
```

13.7.8 Removing all Module Names from the Focus Stack

The command **clear-focus-stack** removes all module names from the focus stack.

Syntax

```
(clear-focus-stack)
```

13.7.9 Setting the Salience Evaluation Behavior

This function sets the salience evaluation behavior. By default, salience values are only evaluated when a rule is defined.

Syntax

```
(set-salience-evaluation <value>)
```

where <value> is either when-defined, when-activated, or every-cycle. The return value for this function is the old value for salience evaluation. The value when-defined forces salience evaluation at the time of rule definition. The value when-activation forces salience evaluation at the time of rule definition and upon being activated. The value every-cycle forces evaluation at the time of rule definition, upon being activated, and after every rule firing.

13.7.10 Getting the Salience Evaluation Behavior

This function returns the current salience evaluation behavior (when-defined, when-activated, or every-cycle).

Syntax

```
(get-salience-evaluation)
```

13.7.11 Refreshing the Salience Value of Rules on the Agenda

This function forces reevaluation of saliences of rules on the agenda regardless of the current salience evaluation setting. This function has no return value.

Syntax

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

If <module-name> is unspecified, then the agenda in the current module is refreshed. If <module-name> is specified, then the agenda in the specified module is refreshed. If <module-name> is the symbol *, then the agenda in every module is refreshed.

13.8 Defglobal Commands

The following commands manipulate defglobals.

13.8.1 Displaying the Text of a Defglobal

Displays the text required to define a given global variable. Note that unlike other constructs such as deffacts and definstances, defglobal definitions have no name associated with the entire construct. The variable name passed to ppdefglobal should not include the question mark or the asterisks (e.g. x is the variable name for the global variable ?*x*). This function has no return value.

Syntax

```
(ppdefglobal <global-variable-name>)
```

13.8.2 Displaying the List of Defglobals

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

Syntax

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

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

13.8.3 Deleting a Defglobal

This function deletes a previously defined defglobal.

Syntax

```
(undefglobal <defglobal-name>)
```

If the defglobal is in use (for example if it is referred to in a deffunction), then the deletion will fail. Otherwise, no further uses of the deleted defglobal are permitted (unless redefined). If the symbol * is used for <defglobal-name>, then all defglobals will be deleted (unless there is a defglobal named *). This function has no return value.

13.8.4 Displaying the Values of Global Variables

Displays the names and current values of all defglobals. This function has no return value.

Syntax

```
(show-defglobals [<module-name>])
```

If <module-name> is unspecified, then the names and values of all defglobals in the current module are displayed. If <module-name> is specified, then the names and values of all defglobals in the specified module are displayed. If <module-name> is the symbol *, then the names and values of all defglobals in all modules are displayed.

13.8.5 Setting the Reset Behavior of Global Variables

This function sets the reset global variables behavior. When this behavior is enabled (TRUE by default) global variables are reset to their original values when the **reset** command is performed. The return value for this function is the old value for the behavior.

Syntax

```
(set-reset-globals <boolean-expression>)
```

13.8.6 Getting the Reset Behavior of Global Variables

This function returns the current value of the reset global variables behavior (TRUE or FALSE).

Syntax

```
(get-reset-globals)
```

13.9 Deffunction Commands

The following commands manipulate deffunctions.

13.9.1 Displaying the Text of a Deffunction

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

Syntax

```
(ppdeffunction <deffunction-name>)
```

13.9.2 Displaying the List of Deffunctions

Displays the names of all deffunctions stored in the CLIPS environment. This function has no return value.

Syntax

```
(list-deffunctions)
```

13.9.3 Deleting a Deffunction

This function deletes a previously defined deffunction.

Syntax

```
(undeffunction <deffunction-name>)
```

If the symbol * is used for <deffunction-name>, then all deffunctions will be deleted (unless there exists a deffunction called *). The undeffunction command can be used to remove deffunctions at any time. Exceptions: A deffunction may not be deleted when it is executing or when there is still a reference to it in another loaded construct, such as a rule RHS. This function has no return value.

13.10 Generic Function Commands

The following commands manipulate generic functions.

13.10.1 Displaying the Text of a Generic Function Header

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

Syntax

```
(ppdefgeneric <generic-function-name>)
```

13.10.2 Displaying the Text of a Generic Function Method

Displays the text of a given method.

Syntax

```
(ppdefmethod <generic-function-name> <index>)
```

where <index> is the method index (see section 8.4.2). This function has no return value.

13.10.3 Displaying the List of Generic Functions

Displays the names of all generic functions stored in the CLIPS environment.

Syntax

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

If <module-name> is unspecified, then the names of all defgenerics in the current module are displayed. If <module-name> is specified, then the names of all defgenerics in the specified module are displayed. If <module-name> is the symbol *, then the names of all defgenerics in all modules are displayed. This function has no return value.

13.10.4 Displaying the List of Methods for a Generic Function

If no name is given, this function lists all generic function methods in the CLIPS environment. If a name is given, then only the methods for the named generic function are listed. The methods are listed in decreasing order of precedence (see section 8.5.2) for each generic function. Method indices can be seen using this function. This function has no return value.

Syntax

```
(list-defmethods [<generic-function-name>])
```

13.10.5 Deleting a Generic Function

This function deletes a previously defined generic function.

Syntax

```
(undefgeneric <generic-function-name>)
```

If the symbol * is used for <generic-function-name>, then all generic functions will be deleted (unless there exists a generic function called *). This function removes the header and all methods for a generic function. The undefgeneric command can be used to remove generic functions at any time. Exceptions: A generic function may not be deleted when any of its methods are executing or when there is still a reference to it in another loaded construct, such as a rule RHS. This function has no return value.

13.10.6 Deleting a Generic Function Method

This function deletes a previously defined generic function method.

Syntax

```
(undefmethod <generic-function-name> <index>)
```

where <index> is the index of the method to be deleted for the generic function. If the symbol * is used for <index>, then all the methods for the generic function will be deleted. (This is different from the undefgeneric command because the header is not removed.) If * is used for <generic-function-name>, then * must also be specified for <index>, and all the methods for all generic functions will be removed. This function removes the specified method for a generic function, but even if the method removed is the last one, the generic function header is not removed. The undefmethod command can be used to remove methods at any time. Exceptions: A method may not be deleted when it or any of the other methods for the same generic function are executing. This function has no return value.

13.10.7 Previewing a Generic Function Call

This debugging function lists all *applicable* methods for a particular generic function call in order of decreasing precedence (see section 8.5.2). The function **list-defmethods** is different in that it lists *all* methods for a generic function.

Syntax

```
(preview-generic <generic-function-name> <expression>*)
```

This function does not actually execute any of the methods, but any side-effects of evaluating the generic function arguments and any query parameter restrictions (see section 8.4.3) in methods do occur. The output for the first example in section 8.5.2 would be as follows:

Example

```
CLIPS> (preview-generic + 4 5)
+ #7  (INTEGER <qry>) (INTEGER <qry>)
+ #8  (INTEGER <qry>) (NUMBER)
+ #3  (INTEGER) (INTEGER)
+ #4  (INTEGER) (NUMBER)
+ #6  (NUMBER) (INTEGER <qry>)
+ #2  (NUMBER) (INTEGER)
+ #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
+ #5  (NUMBER) (NUMBER) ($? PRIMITIVE)
CLIPS>
```

13.11 CLIPS Object-Oriented Language (COOL) Commands

The following commands manipulate elements of COOL.

13.11.1 Class Commands

The following commands manipulate defclasses.

13.11.1.1 Displaying the Text of a Defclass

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

Syntax

```
(ppdefclass <class-name>)
```

13.11.1.2 Displaying the List of Defclasses

Displays the names of all defclasses stored in the CLIPS environment. If <module-name> is unspecified, then the names of all defclasses in the current module are displayed. If <module-name> is specified, then the names of all defclasses in the specified module are displayed. If <module-name> is the symbol *, then the names of all defclasses in all modules are displayed. This function has no return value.

Syntax

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

13.11.1.3 Deleting a Defclass

This function deletes a previously defined defclass and all its subclasses from the CLIPS environment.

Syntax

```
(undefclass <class-name>)
```

If the symbol * is used for <class-name>, then all defclasses will be deleted (unless there exists a defclass called *). The undefclass command can be used to remove defclasses at any time. Exceptions: A defclass may not be deleted if it has any instances or if there is still a reference to it in another loaded construct, such as a generic function method. This function has no return value.

13.11.1.4 Examining a Class

This function provides a verbose description of a class including: abstract role (whether direct instances can be created or not), direct superclasses and subclasses, class precedence list, slots with all their facets and sources, and all recognized message-handlers. This function has no return value.

Syntax

```
(describe-class <class-name>)
```

Example

```
CLIPS>
(defclass CHILD (is-a USER)
  (role abstract)
  (multislot parents (cardinality 2 2))
  (slot age (type INTEGER)
            (range 0 18))
  (slot sex (access read-only)
            (type SYMBOL)
            (allowed-symbols male female)
            (storage shared)))
CLIPS>
(defclass BOY (is-a CHILD)
  (slot sex (source composite)
            (default male)))
CLIPS>
(defmessage-handler BOY play ()
   (println "The boy is now playing..."))
CLIPS> (describe-class CHILD)
================================================================================
********************************************************************************
Abstract: direct instances of this class cannot be created.

Direct Superclasses: USER
Inheritance Precedence: CHILD USER OBJECT
Direct Subclasses: BOY
--------------------------------------------------------------------------------
SLOTS   : FLD DEF PRP ACC STO MCH SRC VIS CRT OVRD-MSG    SOURCE(S)
parents : MLT STC INH RW  LCL RCT EXC PRV RW  put-parents CHILD
age     : SGL STC INH RW  LCL RCT EXC PRV RW  put-age     CHILD
sex     : SGL STC INH  R  SHR RCT EXC PRV  R  NIL         CHILD

Constraint information for slots:

SLOTS   : SYM STR INN INA EXA FTA INT FLT
parents :  +   +   +   +   +   +   +   +  RNG:[-oo..+oo] CRD:[2..2]
age     :                              +      RNG:[0..18]
sex     :  #
--------------------------------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
get-parents primary in class CHILD
put-parents primary in class CHILD
get-age primary in class CHILD
put-age primary in class CHILD
get-sex primary in class CHILD
********************************************************************************
================================================================================
CLIPS>
```

The following table explains the fields and their possible values in the slot descriptions:

Field	Values	Explanation
FLD	SGL/MLT	Field type (single-field or multifield)
DEF	STC/DYN/NIL	Default value (static, dynamic, or none)
PRP	INH/NIL	Propagation to subclasses (inheritable or not inheritable)
ACC	RW/R/INT	Access (read-write, read-only, or initialize-only)
STO	LCL/SHR	Storage (local or shared)
MCH	RCT/NIL	Pattern-match (reactive or non-reactive)
SRC	CMP/EXC	Source type (composite or exclusive)
VIS	PUB/PRV	Visibility (public or private)
CRT	R/W/RW/NIL	Automatically created accessors (read, write, read-write, or none)
OVRD-MSG	<message-name>	Name of message sent for slot-overrides in make-instance, etc.
SOURCE(S)	<class-name>+	Source of slot (more than one class for composite)

In the constraint information summary for the slots, each of the columns shows one of the primitive data types . A + in the column means that any value of that type is allowed in the slot. A # in the column means that some values of that type are allowed in the slot. Range and cardinality constraints are displayed to the far right of each slot's row. The following table explains the abbreviations used in the constraint information summary for the slots:

Abbreviation	Explanation
SYM	Symbol
STR	String
INN	Instance Name
INA	Instance Address
EXA	External Address
FTA	Fact Address
INT	Integer
FLT	Float
RNG	Range
CRD	Cardinality

13.11.1.5 Examining the Class Hierarchy

This function provides a rudimentary display of the inheritance relationships between a class and all its subclasses. Indentation indicates a subclass. Because of multiple inheritance, some classes may appear more than once. Asterisks mark classes which are direct subclasses of more than one class. With no arguments, this function starts with the root class OBJECT. This function has no return value.

Syntax

```
(browse-classes [<class-name>])
```

Example

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
CLIPS> (defclass B (is-a USER))
CLIPS> (defclass C (is-a A B))
CLIPS> (defclass D (is-a USER))
CLIPS> (defclass E (is-a C D))
CLIPS> (defclass F (is-a E))
CLIPS> (browse-classes)
OBJECT
  PRIMITIVE
    NUMBER
      INTEGER
      FLOAT
    LEXEME
      SYMBOL
      STRING
    MULTIFIELD
    ADDRESS
      EXTERNAL-ADDRESS
      FACT-ADDRESS
      INSTANCE-ADDRESS *
    INSTANCE
      INSTANCE-ADDRESS *
      INSTANCE-NAME
  USER
    A
      C *
        E *
          F
    B
      C *
        E *
          F
    D
      E *
        F
CLIPS>
```

13.11.2 Message-handler Commands

The following commands manipulate defmessage-handlers.

13.11.2.1 Displaying the Text of a Defmessage-handler

Displays the text of a given defmessage-handler. This function has no return value.

Syntax

Defaults are in ***bold italics***.

```
(ppdefmessage-handler <class-name> <handler-name>
   [<handler-type>])
<handler-type> ::= around | before | primary | after
```

13.11.2.2 Displaying the List of Defmessage-handlers

With no arguments, this function lists all handlers in the system. With one argument, this function lists all handlers for the specified class. If the optional argument "inherit" is given, inherited message-handlers are also included. This function has no return value.

Syntax

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

Example

List all primary handlers in the system.

```
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
CLIPS> (defmessage-handler A foo ())
CLIPS> (list-defmessage-handlers A)
foo primary in class A
For a total of 1 message-handler.
CLIPS> (list-defmessage-handlers A inherit)
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
foo primary in class A
For a total of 9 message-handlers.
CLIPS>
```

13.11.2.3 Deleting a Defmessage-handler

This function deletes a previously defined message-handler.

Syntax

Defaults are in ***bold italics***.

```
(undefmessage-handler <class-name> <handler-name>
   [<handler-type>])
<handler-type> ::= around | before | primary | after
```

An asterisk can be used to specify a wildcard for any of the arguments. (Wildcards will not work for the class name or handler name if there is a class or handler called *.) The undefmessage-handler command can be used to remove handlers at any time. Exceptions: A handler may not be deleted when it or any of the other handlers for the same class are executing. This function has no return value.

Example

Delete all primary handlers in the system.

```
CLIPS> (undefmessage-handler * *)
CLIPS>
```

13.11.2.4 Previewing a Message

Displays a list of all the applicable message-handlers for a message sent to an instance of a particular class. The level of indentation indicates the number of times a handler is shadowed, and lines connect the beginning and ending portions of the execution of a handler if it encloses shadowed handlers. The right double-angle brackets indicate the beginning of handler execution, and the left double-angle brackets indicate the end of handler execution. Message arguments are not necessary for a preview since they do not dictate handler applicability.

Syntax

```
(preview-send <class-name> <message-name>)
```

Example

For the example in section 9.5.3, the output would be:

```
CLIPS> (preview-send USER my-message)
>> my-message around in class USER
| >> my-message around in class OBJECT
| | >> my-message before in class USER
| | << my-message before in class USER
| | >> my-message before in class OBJECT
| | << my-message before in class OBJECT
| | >> my-message primary in class USER
| | | >> my-message primary in class OBJECT
| | | << my-message primary in class OBJECT
| | << my-message primary in class USER
```

```
| | >> my-message after in class OBJECT
| | << my-message after in class OBJECT
| | >> my-message after in class USER
| | << my-message after in class USER
| << my-message around in class OBJECT
<< my-message around in class USER
CLIPS>
```

13.11.3 Definstances Commands

The following commands manipulate definstances.

13.11.3.1 Displaying the Text of a Definstances

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

Syntax

```
(ppdefinstances <definstances-name>)
```

13.11.3.2 Displaying the List of Definstances

Displays the names of all definstances stored in the CLIPS environment. This function has no return value.

Syntax

```
(list-definstances)
```

13.11.3.3 Deleting a Definstances

This function deletes a previously defined definstances.

Syntax

```
(undefinstances <definstances-name>)
```

If the symbol * is used for <definstances-name>, then all definstances will be deleted (unless there exists a definstances called *). The undefinstances command can be used to remove definstances at any time. Exceptions: A definstances may not be deleted when any of the instances in it are being created. This function has no return value.

13.11.4 Instances Commands

The following commands manipulate instances of user-defined classes.

13.11.4.1 Listing the Instances

If no arguments are specified, all instances in scope of the current module are listed. If a module name is given, all instances within the scope of that module are given. If "*" is specified (and there is no module named "*"), all instances in all modules are listed (only instances which actually belong to classes of a module are listed for each module to prevent duplicates). If a class name is specified, only the instances for the named class are listed. If a class is specified, then the optional keyword *inherit* causes this function to list instances of subclasses of the class as well. This function has no return value.

Syntax

```
(instances [<module-name> [<class-name> [inherit]]])
```

13.11.4.2 Printing an Instance's Slots from a Handler

This function operates implicitly on the active instance (see section 9.4.1.1) for a message, and thus can only be called from within the body of a message-handler. This function directly prints the slots of the active instance and is the one used to implement the print handler attached to class USER (see section 9.4.4.3). This function has no return value.

Syntax

```
(ppinstance)
```

13.11.4.3 Saving Instances to a Text File

This function saves all instances in the CLIPS environment to the specified file in the following format:

```
(<instance-name> of <class-name> <slot-override>*)
<slot-override> ::= (<slot-name> <single-field-value>*)
```

A slot-override is generated for every slot of every instance, regardless of whether the slot currently holds a default value or not. External-address and fact-address slot values are saved as strings. Instance-address slot values are saved as instance-names. This function returns the number of instances saved.

Syntax

```
(save-instances <file-name>
         [local | visible [[inherit] <class>+])
```

By default, save-instances saves only the instances of all defclasses in the current module. Specifying **visible** saves instances for all classes within scope of the current module. Also,

particular classes may be specified for saving, but they must be in scope according to the local or visible option. The inherit keyword can be used to force the saving of indirect instances of named classes as well (by default only direct instances are saved for named classes). Subclasses must still be in local or visible scope in order for their instances to be saved. Unless the inherit option is specified, only concrete classes can be specified. At least one class is required for the inherit option.

The file generated by this function can be loaded by either **load-instances** or **restore-instances**. save-instances does not preserve module information, so the instance file should be loaded into the module which was current when it was saved.

13.11.4.4 Saving Instances to a Binary File

The function **bsave-instances** works exactly like **save-instances** except that the instances are saved in a binary format which can only be loaded with the function **bload-instances**. The advantage to this format is that loading binary instances can be much faster than loading text instances for large numbers of instances. The disadvantage is that the file is not usually portable to other platforms.

Syntax

```
(bsave-instances <file-name>
         [local | visible [[inherit] <class>+])
```

13.11.4.5 Loading Instances from a Text File

This function loads instances from a file into the CLIPS environment. It can read files created with save-instances or any ASCII text file. Each instance should be in the format described in section 13.11.4.3 (although the instance name can be left unspecified). Calling load-instances is exactly equivalent to a series of make-instance calls (in CLIPS version 5.1, slot access restrictions, such as read-only, were suspended during calls to load-instances). This function returns the number of instances loaded or -1 if it could not access the instance file.

Syntax

```
(load-instances <file-name>)
```

13.11.4.6 Loading Instances from a Text File without Message Passing

The function **restore-instances** loads instances from a file into the CLIPS environment. It can read files created with save-instances or any ASCII text file. Each instance should be in the format described in section 13.11.4.3 (although the instance name can be left unspecified). It is similar in operation to load-instances, however, unlike load-instances, restore-instances does not

use message-passing for deletions, initialization, or slot-overrides. Thus in order to preserve object encapsulation, it is recommended that restore-instances only be used with files generated by save-instances. This function returns the number of instances loaded or -1 if it could not access the instance file.

Syntax

```
(restore-instances <file-name>)
```

13.11.4.7 Loading Instances from a Binary File

This function is similar to **restore-instances** except that it can only work with files generated by **bsave-instances**. See section 13.11.4.4 for a discussion of the merits of using binary instance files.

Syntax

```
(bload-instances <file-name>)
```

13.12 Defmodule Commands

The following commands manipulate defmodule constructs.

13.12.1 Displaying the Text of a Defmodule

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

Syntax

```
(ppdefmodule <defmodule-name>)
```

13.12.2 Displaying the List of Defmodules

Displays the names of all defmodule constructs stored in the CLIPS environment. This function has no return value.

Syntax

```
(list-defmodules)
```

13.13 Memory Management Commands

The following commands display CLIPS memory status information. CLIPS memory management is described more fully in the *Advanced Programming Guide*.

13.13.1 Determining the Amount of Memory Used by CLIPS

Returns an integer representing the number of bytes CLIPS has currently in-use or has held for later use. This number does not include operating system overhead for allocating memory.

Syntax

```
(mem-used)
```

13.13.2 Determining the Number of Memory Requests Made by CLIPS

Returns an integer representing the number of times CLIPS has requested memory from the operating system. If the operating system overhead for allocating memory is known, then the total memory used can be calculated by

```
(+ (mem-used) (* <overhead-in-bytes> (mem-requests)))
```

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 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.

Syntax

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

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

13.14 External Text Manipulation

CLIPS provides a set of functions to build and access a hierarchical lookup system for multiple external files. Each file contains a set of text entries in a special format that CLIPS can later reference and display. The basic concept is that CLIPS retains a "map" of the text file in memory and can easily pull sections of text from the file without having to store the whole file in memory and without having to sequentially search the file for the appropriate text.

13.14.1 External Text File Format

Each external text file to be loaded into CLIPS must be described in a particular way. Each topic entry in each file must be in the format shown following.

Syntax

```
<level-num> <entry-type> BEGIN-ENTRY- <topic-name>
          •
          •
Topic information in form to be displayed when referenced.
          •
          •
END-ENTRY
```

The delimiter strings (lines with BEGIN_ENTRY or END_ENTRY info) must be the only things on their lines. Embedded white space between the fields of the delimiters is allowed.

The first parameter, <level-num>, is the level of the hierarchical tree to which the entry belongs. The lower the number, the closer to the root level the topic is; i.e., the lowest level number indicates the root level. Subtopics are indicated by making the level number of the current topic larger than the previous entry (which is to be the parent). Thus, the tree must be entered in the file sequentially; i.e., a topic with all its subtopics must be described before going on to a topic at the same level. Entering a number less than that of the previous topic will cause the tree to be searched upwards until a level number is found which is less than the current one. The current topic then will be attached as a subtopic at that level. In this manner, multiple root trees may be created. Level number and order of entry in a file can indicate the order of precedence in which a list of subtopics that are all children of the same topic will be searched. Topics with the same level number will be searched in the order in which they appear in the file. Topics with lower-level numbers will be searched first.

Example

```
0MBEGIN-ENTRY-ROOT
 --  Text  --
END-ENTRY
2IBEGIN-ENTRY-SUBTOPIC1
 --  Text  --
END-ENTRY
1IBEGIN-ENTRY-SUBTOPIC2
 --  Text  --
END-ENTRY
```

In the above example, SUBTOPIC1 and SUBTOPIC2 are children of ROOT. However, in searching the children of ROOT, SUBTOPIC2 would be found first.

The second parameter in the format defined above, the <entry-type>, must be a single capital letter, either M (for MENU) or I (for INFORMATION). Only MENU entries may have subtopics.

The third parameter defined above, the <topic-name>, can be any alphanumeric string of up to 80 characters. No white space can be embedded in the name.

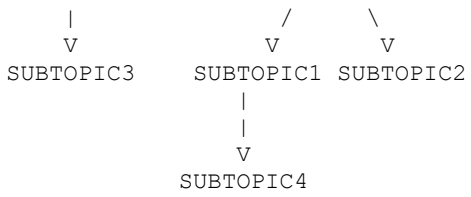
Beginning a line with the delimiter "$$" forces the loader to treat the line as pure text, even if one of the key delimiters is in it. When the line is printed, the dollar signs are treated as blanks.

Example

```
0MBEGIN-ENTRY-ROOT1
 --  Root1 Text  --
END-ENTRY
1MBEGIN-ENTRY-SUBTOPIC1
 --  Subtopic1 Text  --
END-ENTRY
2IBEGIN-ENTRY-SUBTOPIC4
 --  Subtopic4 Text  --
END-ENTRY
1IBEGIN-ENTRY-SUBTOPIC2
 --  Subtopic2 Text  --
END-ENTRY
0IBEGIN-ENTRY-ROOT2
 --  Root2 Text  --
END-ENTRY
-1MBEGIN-ENTRY-ROOT3
 --  Root3 Text  --
END-ENTRY
0IBEGIN-ENTRY-SUBTOPIC3
 --  Subtopic3 Text  --
END-ENTRY
```

Tree Diagram of Above Example :

```
-> ROOT3 ---------> ROOT1 ---------> ROOT2
     |               /   \
```



13.14.2 External Text Manipulation Functions

The following functions can be used by users to maintain their own information system.

13.14.2.1 Fetch

The function **fetch** loads the named file (which must be in the format defined in section 13.14.1) into the internal lookup table.

Syntax

```
(fetch <file-name>)
```

The function returns the number of entries loaded if the fetch succeeded. If the file could not be loaded or was loaded already, the function returns the symbol FALSE.

13.14.2.2 Print-region

The function **print-region** looks up a specified entry in a particular file which has been loaded previously into the lookup table and prints the contents of that entry to the specified output.

Syntax

```
(print-region <logical-name> <file-name> <topic-field>*)
```

where <logical-name> is a name previously attached to an output device. To send the output to stdout, specify t for the logical name. <file-name> is the name of the previously loaded file in which the entry is to be found, and the optional arguments, <topic-field>*, is the full path of the topic entry to be found.

Each element or field in the path is delimited by white space, and the function is not case sensitive. In addition, the entire name of a field does not need to be specified. Only enough characters to distinguish the field from other choices at the same level of the tree are necessary. If there is a conflict, the function will pick the first one in the list. A few special fields can be specified.

^ Branch up one level.

? When specified at the end of a path, this forces a display of the current menu, even on branch-ups.

<nil> Giving no topic field will branch up one level.

The level of the tree for a file remains constant between calls to **print-region**. All levels count from menu only. Information levels do not count for branching up or down. To access an entry at the root level after branching down several levels in a previous call or series of calls, an equal number of branches up must be executed.

Examples

To display the entry for ROOT SUBTOPIC from the file foo.lis on the screen, type

```
(print-region t "foo.lis" ROOT SUBTOPIC)
```

or, using less characters,

```
(print-region t "foo.lis" roo sub)
```

Only one entry can be accessed per **print-region** call. The function returns the symbol TRUE if the print-region succeeded. If the entry was not found, it returns FALSE.

```
CLIPS> (fetch "foo.lis")
7
CLIPS> (print-region t "foo.lis" roo sub)

 --  Subtopic3 Text  --
TRUE
CLIPS> (print-region t "foo.lis" "?")

 --  Root3 Text  --
TRUE
CLIPS> (print-region t "foo.lis" ^ root1 sub)

 --  Subtopic1 Text  --
TRUE
CLIPS> (print-region t "foo.lis" sub)

 --  Subtopic4 Text  --
TRUE
CLIPS> (print-region t "foo.lis" ^ subtopic2)

 --  Subtopic2 Text  --
TRUE
CLIPS> (print-region t "foo.lis" ^ root2)

 --  Root2 Text  --
TRUE
CLIPS> (toss "foo.lis")
TRUE
CLIPS>
```

13.14.2.3 Get-region

The function **get-region** looks up a specified entry in a particular file which has been loaded previously into the lookup table and returns the contents of that entry as a string.

Syntax

```
(get-region <file-name> <topic-field>*)
```

where <file-name> is the name of the previously loaded file in which the entry is to be found, and the optional arguments, <topic-field>*, is the full path of the topic entry to be found. The **get-region** the **print-region** functions share the same behavior for the special topic fields and maintaining the level of the tree for a file between function calls. If an error occurs, this function returns an empty string.

13.14.2.4 Toss

The function **toss** unloads the named file from the internal lookup table and releases the memory back to the system.

Syntax

```
(toss <file-name>)
```

The function returns the symbol TRUE if the toss succeeded. If the file was not on the lookup table, it returns FALSE.

13.15 Profiling Commands

The following commands provide the ability to profile CLIPS programs for performance.

13.15.1 Setting the Profiling Report Threshold

The **set-profile-percent-threshold** command sets the minimum percentage of time that must be spent executing a construct or user function for it to be displayed by the **profile-info** command. By default, the percent threshold is zero, so all constructs or user-functions that were profiled and executed at least once will be displayed by the **profile-info** command. The return value of this function is the old percent threshold.

Syntax

```
(set-profile-percent-threshold <number in the range 0 to 100>)
```

13.15.2 Getting the Profiling Report Threshold

The **get-profile-percent-threshold** command returns the current value of the profile percent threshold.

Syntax

```
(get-profile-percent-threshold)
```

13.15.3 Resetting Profiling Information

The **profile-reset** command resets all profiling information currently collected for constructs and user functions.

Syntax

```
(profile-reset)
```

13.15.4 Displaying Profiling Information

The **profile-info** command displays profiling information currently collected for constructs or user functions. Profiling information is displayed in six columns. The first column contains the name of the construct or user function profiled. The second column indicates the number of times the construct or user function was executed. The third column is the amount of time spent executing the construct or user function. The fourth column is the percentage of time spent in the construct or user function with respect to the total amount of time profiling was enabled. The fifth column is the total amount of time spent in the first execution of the construct or user function and all subsequent calls to other constructs/user functions. The sixth column is the percentage of this time with respect to the total amount of time profiling was enabled.

Syntax

```
(profile-info)
```

13.15.5 Profiling Constructs and User Functions

The **profile** command is used to enable/disable profiling of constructs and user functions. If **constructs** are profiled, then the amount of time spent executing deffunctions, generic functions, message handlers, and the RHS of defrules is tracked. If **user-functions** are profiled, then the time spent executing system and user defined functions is tracked. System defined functions include predefined functions available for your own use such as the **<** and **numberp** functions in addition to low level internal functions which are not available for your use (these will usually appear in **profile-info** output in all capital letters or surrounded by parentheses). It is not possible to profile constructs and user-functions at the same time. Enabling one disables the other. The **off**

keyword argument disables profiling. Profiling can be repeatedly enable and disabled as long as only one of **constructs** or **user-functions** is consistently enabled. The total amount of time spent with profiling enabled will be displayed by the **profile-info** command. If profiling is enabled from the command prompt, it is a good idea to place the calls enabling and disabling profiling within a single **progn** function call. This will prevent the elapsed profiling time from including the amount of time needed to type the commands being profiled.

Syntax

```
(profile constructs | user-functions | off)
```

Example

```
CLIPS> (clear)
CLIPS> (deffacts start (fact 1))
CLIPS>
(deffunction function-1 (?x)
  (bind ?y 1)
  (loop-for-count (* ?x 10)
     (bind ?y (+ ?y ?x))))
CLIPS>
(defrule rule-1
  ?f <- (fact ?x&:(< ?x 100))
  =>
  (function-1 ?x)
  (retract ?f)
  (assert (fact (+ ?x 1))))
CLIPS>
(reset)
CLIPS>
(progn (profile constructs)
       (run)
       (profile off))
CLIPS> (profile-info)
Profile elapsed time = 0.474921 seconds

Construct Name     Entries       Time          %      Time+Kids     %+Kids
--------------     -------      ------       -----    ---------     ------

*** Deffunctions ***

function-1              99      0.436704    91.92%     0.436704     91.92%

*** Defrules ***

rule-1                  99      0.027561     5.80%     0.464265     97.72%
CLIPS> (profile-reset)
CLIPS> (reset)
CLIPS>
(progn (profile user-functions)
       (run)
       (profile off))
CLIPS> (profile-info)
Profile elapsed time = 12.0454 seconds

Function Name      Entries       Time          %      Time+Kids     %+Kids
-------------      -------      ------       -----    ---------     ------
retract                 99      0.007953     0.07%     0.010646      0.09%
```

```
assert                    99      0.012160     0.10%     0.032766      0.27%
run                        1      0.047421     0.39%    12.045301    100.00%
profile                    1      0.000049     0.00%     0.000049      0.00%
*                         99      0.005579     0.05%     0.007610      0.06%
+                      49599      3.626217    30.10%     5.765490     47.86%
<                         99      0.005234     0.04%     0.007749      0.06%
progn                  49698      2.353003    19.53%    11.997880     99.61%
loop-for-count            99      1.481078    12.30%    11.910553     98.88%
PCALL                     99      0.020747     0.17%    11.943234     99.15%
FACT_PN_VAR3              99      0.002515     0.02%     0.002515      0.02%
FACT_JN_VAR1              99      0.002693     0.02%     0.002693      0.02%
FACT_JN_VAR3             198      0.004718     0.04%     0.004718      0.04%
FACT_STORE_MULTIFIELD     99      0.005478     0.05%     0.012857      0.11%
PROC_PARAM             49599      1.036460     8.60%     1.036460      8.60%
PROC_GET_BIND          49500      1.102682     9.15%     1.102682      9.15%
PROC_BIND              49599      2.331363    19.35%     8.089474     67.16%
CLIPS> (set-profile-percent-threshold 1)
0.0
CLIPS> (profile-info)
Profile elapsed time = 12.0454 seconds

Function Name      Entries       Time          %      Time+Kids     %+Kids
-------------      -------      ------       -----    ---------     ------
+                    49599      3.626217    30.10%     5.765490     47.86%
progn                49698      2.353003    19.53%    11.997880     99.61%
loop-for-count          99      1.481078    12.30%    11.910553     98.88%
PROC_PARAM           49599      1.036460     8.60%     1.036460      8.60%
PROC_GET_BIND        49500      1.102682     9.15%     1.102682      9.15%
PROC_BIND            49599      2.331363    19.35%     8.089474     67.16%
CLIPS> (profile-reset)
CLIPS> (profile-info)
CLIPS>
```

Appendix A:
Support Information

A.1 Questions and Information

The URL for the CLIPS Web page is http://www.clipsrules.net.

Questions regarding CLIPS can be posted to one of several online forums including the CLIPS Expert System Group, http://groups.google.com/group/CLIPSESG/, the SourceForge CLIPS Forums, http://sourceforge.net/forum/?group_id=215471, and Stack Overflow, http://stackoverflow.com/questions/tagged/clips.

Inquiries related to the use or installation of CLIPS can be sent via electronic mail to support@clipsrules.net.

A.2 Documentation

The CLIPS Reference Manuals and other documentation is available at http://www.clipsrules.net/?q=Documentation.

Expert Systems: Principles and Programming, 4th Edition, by Giarratano and Riley comes with a CD-ROM containing CLIPS 6.22 executables (DOS, Windows XP, and Mac OS), documentation, and source code. The first half of the book is theory oriented and the second half covers rule-based, procedural, and object-oriented programming using CLIPS.

A.3 CLIPS Source Code and Executables

CLIPS executables and source code are available on the SourceForge web site at http://sourceforge.net/projects/clipsrules/files.

Appendix B:
Update Release Notes

The following sections denote the changes and bug fixes for CLIPS versions 6.3 and 6.4.

B.1 Version 6.40

- **Initial Fact** – The initial-fact deftemplate and deffacts are no longer supported.
- **Initial Object** – The INITIAL-OBJECT defclass and initial-object definstances are no longer supported.
- **New Functions and Commands** - Several new functions and commands have been added. They are:
  - **print** (see section 12.4.2.3)
  - **println** (see section 12.4.2.3)
  - **unget-char** (see section 12.4.2.10)
  - **flush** (see section 12.4.2.13)
  - **rewind** (see section 12.4.2.14)
  - **tell** (see section 12.4.2.15)
  - **seek** (see section 12.4.2.16)
  - **local-time** (see section 12.7.12)
  - **gm-time** (see section 12.7.13)
  - **get-error** (see section 12.7.14)
  - **clear-error** (see section 12.7.15)
  - **set-error** (see section 12.7.16)
- **Command and Function Changes** - The following commands and functions have been changed:

- **assert** (see section 12.9.1). When a duplicate fact is asserted, the return value of the **assert** command is the originally asserted fact. The symbol **false** is only returned by the **assert** command if an error occurs.

- **duplicate** (see section 12.9.4). The return value of a function call can be used to specify the fact being duplicated. Specifying the fact using a fact-index is no longer limited to top-level commands.

- **eval** (see section 12.3.5). When executed from the command prompt, the eval function can access previously bound local variables. The eval function is now available in binary-load only and run-time CLIPS configurations.

- **funcall** (see section 12.7.9). A module specifier can be used as part of the function name when referencing a deffunction or defgeneric that is exported by a module.

- **open** (see section 12.4.2.1). The r+, w+, and a+ modes and their binary counterparts are now supported.

- **load** (see section 13.1.1). The file name and line number are now printed for each error/warning message generated during execution of this command.

- **modify** (see section 12.9.3). The **modify** command now preserves the fact-index and fact-address of the fact being modified. Modifying a fact without changing any slots no longer retracts and reasserts the original fact. If facts are being watched, only changed slots are displayed when a fact is being modified. The return value of a function call can be used to specify the fact being modified. Specifying the fact using a fact-index is no longer limited to top-level commands. If all slot changes specified in the modify command match the current values of the fact to be modified, no action is taken.

- **read** (see section 12.4.2.4). The **read** function now returns symbols for tokens that are not primitive values. For example, the token ?var is returned as the symbol ?var and not the string "?var".

- **system** (see section 13.1.12). The **system** function now returns an integer completion status.

- **str-index** (see section 12.3.4). The **str-index** function now returns 1 if the search string is the empty string "".

- **watch** (see section 13.2.3). The compilations watch flag now defaults to off.

- **Incremental Reset** – This behavior is now always enabled—newly defined rules are always updated based upon the current state of the fact-list. The functions **get-incremental-reset** and **set-incremental-reset** are no longer supported.

- **Static Constraint Checking** – This behavior is now always enabled—constraint violations are always checked when function calls and constructs are parsed. The functions **get-static-constraint-checking** and **set-static-constraint-checking** are no longer supported.

- **Auto Float Dividend** – This behavior is now always enabled— the dividend of the division function is always automatically converted to a floating point number. The functions **get-auto-float-dividend** and **set-auto-float-dividend** are no longer supported.

- **Legacy Functions** – The functions **direct-mv-delete**, **direct-mv-insert**, **direct-mv-replace**, **length**, **member**, **mv-append**, **mv-delete**, **mv-delete**, **mv-slot-delete**, **mv-slot-insert**, **mv-slot-replace**, **mv-subseq**, **nth**, **str-explode**, **str-implode**, and **subset** are no longer supported.

- **Fact Query Pruning** – The fact set query functions (see section 12.9.12) now prune all fact sets containing fact retracted by actions applied to prior fact sets.

- **Instance Query Pruning** – The instance set query functions (see sections 9.7) now prune all instance sets containing instances deleted by actions applied to prior instance sets.

- **Retracted Fact Errors** – The following functions now generate errors when used with retracted facts: **dependencies**, **dependents**, **duplicate**, **fact-index**, **fact-relation**, **fact-slot-names**, **fact-slot-value**, **modify**, **ppfact**, and **timetag**.

- **Logical Names** – The **wclips**, **wdialog**, **wdisplay**, and **wtrace** logical names are no longer supported. Output previously directed to these logical names is now sent to **stdout**.

B.2 Version 6.30

- **Performance Improvements** – Rule performance has been improved particularly in situations with large numbers of fact/instances or partial matches.

- **64-bit Integers** – Integers in CLIPS are now represented using the “long long” C data type which provides a minimum of 64 bits of precision.

- **Reset after Clear** – A reset command is now performed after a clear command (which includes the clear command issued internally by CLIPS when it is started). Since no user constructs will be present after a clear, the primary effect of this behavior is to create the initial-fact and initial-object.

- **Pattern Addition** – The initial-fact and initial-object patterns are no longer used in triggering rules. When printing partial matches, the * symbol is used to indicate a not or exists pattern that is satisfied.

- **Module Specifiers** – A module specifier can be used in expressions to reference a deffunction or defgeneric that is exported by a module, but not specifically imported by the module which is referencing it. For example: (UTIL::my-function a 3).

- **Instance Name and Class Visibility** – Instance names now have global scope and must be unique regardless of their module. Instances of classes that are not in scope can be created if the module name is specified as part of the class name. Messages can be sent to instances regardless of whether the instance class is in scope.

- **Command Prompt** – Local variables bound at the command prompt using the bind function persist until a reset or clear command is issued (see section 2.1.1).

- **Printout Function** – The deprecated use of the symbol *t* as a substitute for the *crlf* symbol is no longer allowed.

- **MicroEMACS Editor** – The built-in editor is no longer supported.

- **New Functions and Commands** - Several new functions and commands have been added. They are:

  - **foreach** (see section 12.6.10)

  - **operating-system** (see section 12.7.11)

- **Command and Function Changes** - The following commands and functions have been enhanced:

  - **matches** (see section 13.6.4). This command now has a return value indicating the number of matches, partial matches, and activations. The amount of output can be controlled with a verbosity argument.

  - **open** (see section 12.4.2.1). The r+ mode is no longer supported. New modes ab and rb have been added.

- **Help Functions** – The **help** and **help-path** funtions are no longer supported

- **Behavior Changes** - The following changes have been made to behavior:

  - A defgeneric redefinition warning is no longer printed when a defmethod is defined.

Appendix C: Glossary

This section defines some of the terminology used throughout this manual.

abstraction The definition of new classes to describe the common properties and behavior of a group of objects.

action A function executed by a construct (such as the RHS of a rule) which typically has no return value, but performs some useful action (such as the printout action) (see section 12).

activation A rule is activated if all of its conditional elements are satisfied and it has not yet fired based on a specific set of matching pattern entities that caused it to be activated. Note that a rule can be activated by more than one set of pattern entities. An activated rule that is placed on the agenda is called an activation.

active instance The object responding to a message which can be referred to by ?self in the message's handlers.

agenda A list of all rules that are presently ready to fire. It is sorted by salience values and the current conflict resolution strategy. The rule at the top of the agenda is the next rule that will fire.

antecedent The LHS of a rule.

bind The action of storing a value in a variable.

class Template for describing the common properties (slots) and behavior (message-handlers) of a group of objects called instances of the class.

class precedence list A linear ordering of classes which describes the path of inheritance for a class.

command A function executed at the top-level command prompt (such as the reset command) typically having no return value.

command prompt In the interactive interface, the "CLIPS>" prompt which indicates that CLIPS is ready for a command to be entered.

condition A conditional element.

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.

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 deftemplate object is simply an informal term for the collections of slots (without specific values) which define a deftemplate. Deftemplate objects do not have inheritance

deftemplate pattern A list of named constraints (constrained slots). A deftemplate pattern describes the attributes and associated values of a deftemplate object. Also called a non-ordered pattern.

delimiter A character which indicates the end of a symbol. 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 parenthesis "(" and ")", an ampersand "&", a vertical bar "|", a less than "<", a semicolon ";", and a tilde "~".

dynamic binding The deferral of which message-handlers will be called for a message until run-time.

encapsulation The requirement that all manipulation of instances of user-defined classes be done with messages.

expression A function call with arguments specified.

external-address 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 (see section 2.3.1 for more details).

external function A function written in an external language (such as C or Ada) defined by the user or provided by CLIPS and called from within CLIPS rules.

facet A component of a slot specification for a class, e.g. default value and cardinality.

fact An ordered or deftemplate (non-ordered) fact. Facts are the data about which rules reason and represent the current state of the world.

fact-address A pointer to a fact obtained by binding a variable to the fact which matches a pattern on the LHS of a rule.

fact-identifier A shorthand notation for referring to a fact. It consists of the 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.

inheritance The process whereby one class can be defined in terms of other class(es).

instance An object is an instance of a class. Throughout the documentation, the term instance usually refers to objects which are instances of user-defined classes.

instance (of a user-defined class) An object which can only be manipulated via messages, i.e all objects except symbols, strings, integers, floats, multifields and external-addresses.

instance-address The address of an instance of a user-defined class (see section 2.3.1 for more details).

instance-name A symbol enclosed within left and right brackets (see section 2.3.1 for more details). An instance-name refers to an object of the specified name which is an instance of a user-defined class.

instance-set An ordered collection of instances of user-defined classes. Each member of an instance-set is an instance of a set of classes, where the set can be different for each member.

instance-set distributed action A user-defined expression which is evaluated for every instance-set which satisfies an instance-set query.

instance-set query A user-defined boolean expression applied to an instance-set to see if it satisfies further user-defined criteria.

integer A number that begins with an optional sign followed by one or more digits (see section 2.3.1 for more details).

LHS Left-Hand Side. The set of conditional elements that must be satisfied for the actions of the RHS of a rule to be performed.

list A group of items with no implied order.

logical name A symbolic name that is associated with an I/O source or

destination.

message The mechanism used to manipulate an object.

message dispatch The process whereby applicable message-handlers are selected and executed for a particular message.

message-handler An implementation of a message for a particular class of objects.

message-handler precedence The property used by the message dispatch to select between handlers when more than one is applicable to a particular message.

method An implementation of a generic function for a particular set of argument restrictions.

method index A shorthand notation for referring to a method with a particular set of parameter restrictions.

method precedence The property used by the generic dispatch to select a method when more than one is applicable to a particular generic function call.

module A workspace where a set of constructs can be grouped together such that explicit control can be maintained over restricting the access of the constructs by other modules. Also used to control the flow of execution of rules through the use of the focus command.

module specifier A notation for specifying a module. It consists of a module name followed by two colons. When placed before a construct name, it's used to specify which module a newly defined construct is to be added to or to specify which construct a command will affect if that construct is not in the current module.

multifield A sequence of unnamed placeholders each having a value.

multifield value A sequence of zero or more single-field values.

non-ordered fact A deftemplate fact.

number An integer or float.

object A symbol, a string, a floating-point or integer number, a multifield value, an external address or an instance of a user-defined class.

order	Position is significant.
ordered fact	A sequence of unnamed fields.
ordered pattern	A sequence of constraints.
overload	The process whereby a generic function can do different things depending on the types and number of its arguments, i.e. the generic function has multiple methods.
pattern	A conditional element on the LHS of a rule which is used to match facts in the fact-list.
pattern entity	An item that is capable of matching a pattern on the LHS of a rule. Facts and instances are the only types of pattern entities available.
pattern-matching	The process of matching facts or instances to patterns on the LHS of rules.
polymorphism	The ability of different objects to respond to the same message in a specialized manner.
primitive type object	A symbol, string, integer, float, multifield or external-address.
relation	The first field in a fact or fact pattern. Synonomous with the associated deftemplate name.
RHS	Right-Hand Side. The actions to be performed when the LHS of a rule is satisfied.
rule	A collection of conditions and actions. When all patterns are satisfied, the actions will be taken.
salience	A priority number given to a rule. When multiple rules are ready for firing, they are fired in order of priority. The default salience is zero (0). Rules with the same salience are fired according to the current conflict resolution strategy.
sequence	An ordered list.
shadowed message-handler	A message-handler that must be explicitly called by another message-handler in order to execute.

shadowed method A method that must be explicitly called by another method in order to execute.

single-field value One of the primitive data types: float, integer, symbol, string, external-address, instance-name, or instance-address.

slot Named single-field or multifield. To write a slot give the field name (attribute) followed by the field value. A single-field slot has one value, while a multifield slot has zero or more values. Note that a multifield slot with one value is strictly not the same as a single field slot. However, the value of a single-field slot (or variable) may match a multifield slot (or multifield variable) that has one field.

slot-accessor Implicit message-handlers which provide read and write access to slots of an object.

specificity (class) A class that precedes another class in a class precedence list is said to be more specific. A class is more specific than any of its superclasses.

specificity (rule) A measure of how "specific" the LHS of a rule is in the pattern-matching process. The specificity is determined by the number of constants, variables, and function calls used within LHS conditional elements.

string A set of characters that starts with double quotes (") and is followed by zero or more printable characters and ends with double quotes (see section 2.3.1 for more details).

subclass If a class inherits from a second class, the first class is a subclass of the second class.

superclass If a class inherits from a second class, the second class is a superclass of the first class.

symbol Any sequence of characters that starts with any printable ASCII character and is followed by zero or more characters (see section 2.3.1 for more details).

top-level In the interactive interface, the "CLIPS>" prompt which indicates that CLIPS is ready for a command to be entered.

value	A single or multifield value.
variable	An symbolic location which can store a value.

Appendix D:
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.

D.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.

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 the rest of the rule will not be considered until that pattern exists. This is most effective if the single pattern consists only of literal constraints. If multiple patterns with variable bindings control rule firing, arrange the patterns so the most important variables are bound first and compared as soon as possible to the other pattern constraints. The use of phase facts is not recommended for large programs if they are used solely for controlling the flow of execution (use modules instead).

2) Patterns with the lowest number of occurrences in the fact-list or instance-list should go near the top. A large number of patterns of a particular form in the fact-list or instance-list can cause numerous partial instantiations of a rule that have to be "weeded" out by comparing the variable bindings, a slower operation.

3) Volatile patterns (ones that are retracted and asserted continuously) should go last, particularly if the rest of the patterns are mostly independent. Every time a pattern entity is created, it must be filtered through the network. If a pattern entity causes a partial rule instantiation, the variable bindings must be considered. By putting volatile patterns last, the variable bindings only will be checked if all of the rest of the patterns already exist.

These rules are *not* independent and commonly conflict with each other. At best, they provide some rough guidelines. Since all systems have these characteristics in different proportions, at a glance the most efficient manner of ordering patterns for a given system is not evident. The best approach is to develop the rules with minimal consideration of ordering. When the reasoning is fairly well verified, experiment with the patterns until the optimum configuration is found.

Another performance issue is the use of multifield variables and wildcards ($?). Although they provide a powerful capability, they must be used very carefully. Since they can bind to zero or more fields, they can cause multiple instantiations of a single rule. In particular, the use of multiple multifield variables in one pattern can cause a very large number of instantiations.

Some final notes on rule performance. Experience suggests that the user should keep the expert system "lean and mean." The list of pattern entities should not be used as a data base for storage of extraneous information. Store and pattern-match only on that information necessary for reasoning. Keep the pattern-matching to a minimum and be as specific as possible. Many short, simple rules perform better than long, complex rules and have the added benefit of being easier to understand and maintain.

D.2 Deffunctions versus Generic Functions

Deffunctions execute more quickly than generic function because generic functions must first examine their arguments to determine which methods are applicable. If a generic function has only one method, a deffunction probably would be better. Care should be taken when determining if a particular function truly needs to be overloaded. In addition, if recompiling and relinking CLIPS is not prohibitive, user-defined external functions are even more efficient than deffunctions. This is because deffunction are interpreted whereas external functions are directly executed. For more details, see sections 7 and 8.2.

D.3 Ordering of Method Parameter Restrictions

When the generic dispatch examines a generic function's method to determine if it is applicable to a particular set of arguments, it examines that method's parameter restrictions from left to right. The programmer can take advantage of this by placing parameter restrictions which are less frequently satisfied than others first in the list. Thus, the generic dispatch can conclude as quickly as possible when a method is not applicable to a generic function call. If a group of restrictions are all equally likely to be satisfied, placing the simpler restrictions first, such as those without queries, will also allow the generic dispatch to conclude more quickly for a method that is not applicable. For more details, see section 8.4.3.

D.4 Instance-Addresses versus Instance-Names

COOL allows instances of user-defined classes to be referenced either by address or by name in functions which manipulate instances, such as message-passing with the **send** function. However, when an instance is referenced by name, CLIPS must perform an internal lookup to find the instance-address anyway. If the same instance is going to be manipulated many times, it might be advantageous to store the instance-address and use that as a reference. This will allow CLIPS to always go directly to the instance. For more details, see sections 2.4.2 and 12.16.4.6.

D.5 Reading Instance Slots Directly

Normally, message-passing must be used to read or set a slot of an instance. However, slots can be read directly within instance-set queries and message-handlers, and they can be set directly within message-handlers. Accessing slots directly is significantly faster than message-passing. Unless message-passing is required (because of slot daemons), direct access should be used when allowed. For more details, see sections 9.4.2, 9.4.3, 9.4.4, 9.6.3, 9.6.4 and 9.7.3.

Appendix E:
CLIPS Warning Messages

CLIPS typically will display two kinds of warning messages: those associated with executing constructs and those associated with loading constructs. This appendix describes some of the more common warning messages and what they mean. Each message begins with a unique identifier enclosed in brackets followed by the keyword **WARNING**; the messages are listed here in alphabetic order according to the identifier.

[CSTRCPSR1] WARNING: Redefining <constructType>: <constructName>
or
[CSTRCPSR1] WARNING: Method # <method index> redefined.
This indicates that a previously defined construct of the specified type has been redefined.

[CSTRNBIN1] WARNING: Constraints are not saved with a binary image when dynamic constraint checking is disabled
or
[CSTRNCMP1] WARNING: Constraints are not saved with a constructs-to-c image when dynamic constraint checking is disabled
These warnings occur when dynamic constraint checking is disabled and the **constructs-to-c** or **bsave** commands are executed. Constraints attached to deftemplate and defclass slots will not be saved with the runtime or binary image in these cases since it is assumed that dynamic constraint checking is not required. Enable dynamic constraint checking with the **set-dynamic-constraint-checking** function before calling **constructs-to-c** or **bsave** in order to include constraints in the runtime or binary image.

[DFFNXFUN1] WARNING: Deffunction <name> only partially deleted due to usage by other constructs.
During a clear or deletion of all deffunctions, only the actions of a deffunction were deleted because another construct which also could not be deleted referenced the deffunction.

Example:

```
CLIPS>
(deffunction foo ()
  (println "Hi there!"))
CLIPS>
(deffunction bar ()
  (foo)
  (undeffunction *))
CLIPS> (bar)
```

[GENRCBIN1] WARNING: COOL not installed! User-defined class in method restriction substituted with OBJECT.
This warning occurs when a generic function method restricted by defclasses is loaded using the **bload** command into a CLIPS configuration where the object language is not enabled. The restriction containing the defclass will match any of the primitive types.

TBD
PRCCODE,4 Execution halted…
SCANNER,1 Over or underflow of long long integer

Appendix F:
CLIPS Error Messages

CLIPS typically will display two kinds of error messages: those associated with executing constructs and those associated with loading constructs. This appendix describes some of the more common error messages and what they mean. Each message begins with a unique identifier enclosed in brackets; the messages are listed here in alphabetic order according to the identifier.

[ANALYSIS1] Duplicate pattern-address <variable name> found in CE <CE number>.
This message occurs when two facts or instances are bound to the same pattern-address variable.

Example:
```
CLIPS> (defrule error ?f <- (a) ?f <- (b) =>)
```

[ANALYSIS2] Pattern-address <variable name> used in CE #2 was previously bound within a pattern CE.
A variable first bound within a pattern cannot be later bound to a fact-address.

Example:
```
CLIPS> (defrule error (a ?f) ?f <- (b) =>)
```

[ANALYSIS3] Variable <variable name> is used as both a single and multifield variable in the LHS.
Variables on the LHS of a rule cannot be bound to both single and multifield variables.

Example:
```
CLIPS> (defrule error (a ?x $?x) =>)
```

[ANALYSIS4] Variable <variable name> [found in the expression <expression>] was referenced in CE <CE number> <field or slot identifier> before being defined
A variable cannot be referenced before it is defined and, thus, results in this error message.

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

[ARGACCES1] 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.

[ARGACCES1] 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.

[ARGACCES1] 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.

[ARGACCES2] Function <name> expected argument #<number> to be of type <data-type>.
This error occurs when a function is passed the wrong type of argument.

[ARGACCES3] Function <function-name> was unable to open file <file-name>.
This error occurs when the specified function cannot open a file.

[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.

[BLOAD3] File <file-name> is an incompatible binary construct file.
This error occurs when the bload command is used to load a file that was created with the bsave command using a different version of CLIPS.

[BLOAD4] The CLIPS environment could not be cleared.
Binary load cannot continue.
A binary load cannot be performed unless the current CLIPS environment can be cleared.

[BLOAD5] Some constructs are still in use by the current binary image:
<construct-name 1>
<construct-name 2>
...
<construct-name N>
Binary <operation> cannot continue.
This error occurs when the current binary image cannot be cleared because some constructs are still being used. The <operation> in progress may either be a binary load or a binary clear.

[BLOAD6] The following undefined functions are referenced by this binary image:
<function-name 1>
<function-name 2>
...
<function-name N>
This error occurs when a binary image is loaded that calls functions which were available in the CLIPS executable that originally created the binary image, but which are not available in the CLIPS executable that is loading the binary image.

[BSAVE1] Cannot perform a binary save while a binary load is in effect.
The bsave command does not work when a binary image is loaded.

[CLASSEXM1] Inherited slot <slot-name> from class <class-name> is not valid for function <name>.
This error message occurs when functions expecting a slot name defined for a class is given an inherited slot.

Example:

```
CLIPS>
(defclass FOO (is-a USER)
  (slot woz (visibility private)))
CLIPS>
(defclass BAR (is-a FOO))
CLIPS> (slot-publicp BAR woz)
```

[CLASSFUN1] Unable to find class <class name> in function <function name>.
This error message occurs when a function is given a non-existent class name.

Example:

```
CLIPS> (class-slots FOO)
```

[CLASSFUN2] Maximum number of simultaneous class hierarchy traversals exceeded <number>.
This error is usually caused by too many simultaneously active instance-set queries, e.g., **do-for-all-instances**. The direct or indirect nesting of instance-set query functions is limited in the following way:

C_i is the number of members in an instance-set for the ith nested instance-set query function.

N is the number of nested instance-set query functions.

$$\sum_{i=1}^{N} C_i \quad <= 128 \text{ (the default upper limit)}$$

Example:

```
CLIPS>
(deffunction my-func ()
  (do-for-instance ((?a USER) (?b USER) (?c USER)) TRUE
     (println ?a " " ?b " " ?c))
; The sum here is C1 = 3 which is OK.
CLIPS>
(do-for-all-instances ((?a OBJECT) (?b OBJECT)) TRUE
   (my-func))

; The sum here is C1 + C2 = 2 + 3 = 5 which is OK.
```

The default upper limit of 128 should be sufficient for most if not all applications. However, the limit may be increased by editing the header file OBJECT.H and recompiling CLIPS.

[CLASSPSR1] An abstract class cannot be reactive.
Only concrete classes can be reactive.

Example:
```
CLIPS>
(defclass FOO (is-a USER)
              (role abstract)
              (pattern-match reactive))
```

[CLASSPSR2] Cannot redefine a predefined system class.
Predefined system classes cannot be modified by the user.

Example:
```
CLIPS> (defclass STRING (is-a NUMBER))
```

[CLASSPSR3] Class <name> cannot be redefined while outstanding references to it still exist.
This error occurs when an attempt to redefine a class is made under one or both of the following two circumstances:

1) The class (or any of its subclasses) has instances.

2) The class (or any of its subclasses) appear in the parameter restrictions of any generic function method.

Before the class can be redefined, all such instances and methods must be deleted.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (defmethod foo ((?a A LEXEME)))
CLIPS> (defclass A (is-a OBJECT)))
```

[CLASSPSR4] The <attribute> class attribute is already declared.
Only one specification of a class attribute is allowed.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (role abstract)
  (role concrete))
```

[CLSLTPSR1] The <slot-name> slot for class <class-name> is already specified.
Slots in a defclass must be unique.

Example:
```
CLIPS>
```

```
(defclass A (is-a USER)
  (slot foo)
  (slot foo))
```

[CLSLTPSR2] The <name> facet for slot <slot-name> is already specified.
Only one occurrence of a facet per slot is allowed.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo (access read-only)
            (access read-write)))
```

[CLSLTPSR3] The 'cardinality' facet can only be used with multifield slots.
Single-field slots by definition have a cardinality of one.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo (cardinality 3 5)))
```

[CLSLTPSR4] Slots with an 'access' facet value of 'read-only' must have a default value.
Since slots cannot be unbound and **read-only** slots cannot be set after initial creation of the instance, **read-only** slots must have a default value.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo (access read-only)
            (default ?NONE)))
```

[CLSLTPSR5] Slots with an 'access' facet value of 'read-only' cannot have a write accessor.
Since **read-only** slots cannot be changed after initializationof the instance, a **write** accessor (**put-** message-handler) is not allowed.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo (access read-only)
            (create-accessor write)))
```

[CLSLTPSR6] Slots with a 'propagation' value of 'no-inherit' cannot have a 'visibility' facet value of 'public'.
no-inherit slots are by definition not accessible to subclasses and thus only visible to the parent class.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo (propagation no-inherit)
            (visibility public)))
```

[COMMLINE1] Expected a '(', constant, or global variable.
This message occurs when a top-level command does not begin with a '(', constant, or global variable.

Example:
```
CLIPS> )
```

[COMMLINE2] Expected a command.
This message occurs when a top-level command is not a symbol.

Example:
```
CLIPS> ("facts"
```

[CONSCOMP1] Invalid file name <fileName> contains '.'
A '.' cannot be used in the file name prefix that is passed to the constructs-to-c command since this prefix is used to generate file names and some operating systems do not allow more than one '.' to appear in a file name.

[CONSCOMP2] Aborting because the base file name may cause the fopen maximum of <integer> to be violated when file names are generated.
The constructs-to-c command generates file names using the file name prefix supplied as an argument. If this base file name is longer than the maximum supported by the operating system, then the possibility exists that files may be overwritten.

[CONSTRCT1] Some constructs are still in use. Clear cannot continue.
This error occurs when the clear command is issued when a construct is in use (such as a rule that is firing).

[CSTRCPSR1] Expected the beginning of a construct.
This error occurs when the load command expects a left parenthesis followed a construct type and these token types are not found.

[CSTRCPSR2] Missing name for <construct-type> construct.
This error occurs when the name is missing for a construct that requires a name.

Example:
```
CLIPS> (defgeneric ())
```

[CSTRCPSR3] Cannot define <construct-type> <construct-name> because of an import/export conflict.
or
[CSTRCPSR3] Cannot define defmodule <defmodule-name> because of an import/export conflict cause by the <construct-type> <construct-name>.
A construct cannot be defined if defining the construct would allow two different definitions of the same construct type and name to both be visible to any module.

Example:
```
CLIPS> (defmodule MAIN (export ?ALL))
CLIPS> (deftemplate MAIN::foo)
CLIPS> (defmodule BAR (import MAIN ?ALL))
CLIPS> (deftemplate BAR::foo (slot x))
```

[CSTRCPSR4] Cannot redefine <construct-type> <construct-name> while it is in use.
A construct cannot be redefined while it is being used by another construct or other data structure (such as a fact or instance).

Example:
```
CLIPS> (clear)
CLIPS> (deftemplate bar)
CLIPS> (assert (bar))
<Fact-0>
CLIPS> (deftemplate (bar (slot x)))
```

[CSTRNCHK1] ***Message Varies***
This error ID covers a range of messages indicating a type, value, range, or cardinality violation.

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.

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)))
```

[CSTRNPSR6] Minimum 'cardinality' value must be greater than or equal to zero.
A multislot with no value has a cardinality of 0. It is not possible to have a lower cardinality.

Example:
```
CLIPS> (deftemplate foo (multislot x (cardinality -3 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)))
```

[DFFNXPSR1] Deffunctions are not allowed to replace constructs.
A deffunction cannot have the same name as any construct.

Example:
```
CLIPS> (deffunction defgeneric ())
```

[DFFNXPSR2] Deffunctions are not allowed to replace external functions.
A deffunction cannot have the same name as any system or user-defined external function.

Example:
```
CLIPS> (deffunction + ())
```

[DFFNXPSR3] Deffunctions are not allowed to replace generic functions.
A deffunction cannot have the same name as any generic function.

Example:
```
CLIPS> (defgeneric foo)
CLIPS> (deffunction foo ())
```

[DFFNXPSR4] Deffunction <name> may not be redefined while it is executing.
A deffunction can be loaded at any time except when a deffunction of the same name is already executing.

Example:
```
CLIPS>
(deffunction foo ()
  (build "(deffunction foo ())"))
CLIPS> (foo)
```

[DFFNXPSR5] Defgeneric <name> imported from module <module name> conflicts with this deffunction.
A deffunction cannot have the same name as any generic function imported from another module.

Example:
```
CLIPS> (defmodule MAIN (export ?ALL))
CLIPS> (defmethod foo ())
CLIPS> (defmodule FOO (import MAIN ?ALL))
CLIPS> (deffunction foo)
```

[DRIVE1] This error occurred in the join network.
Problem resides in associated join
Of pattern #<pattern-number> in rule <rule-name>
This error pinpoints other evaluation errors associated with evaluating an expression within the join network. The specific pattern of the problem rules is identified.

[EMATHFUN1] Domain error for <function-name> function.
This error occurs when an argument passed to a math function is not in the domain of values for which a return value exists.

[EMATHFUN2] Argument overflow for <function-name> function.
This error occurs when an argument to an extended math function would cause a numeric overflow.

[EMATHFUN3] Singularity at asymptote in <function-name> function.
This error occurs when an argument to a trigonometric math function would cause a singularity.

[EVALUATN1] Variable <name> is unbound
This error occurs when a local variable not set by a previous call to **bind** is accessed at the top-level.

Example:
```
CLIPS> (progn ?error)
```

[EXPRNPSR1] A function name must be a symbol.
In the following example, '~' is recognized by CLIPS as an operator, not a function:

Example:
```
CLIPS> (+ (~ 3 4) 4)
```

[EXPRNPSR2] Expected a constant, variable, or expression.
In the following example, '~' is an operator and is illegal as an argument to a function call:

Example:
```
CLIPS> (<= ~ 4)
```

[EXPRNPSR3] Missing function declaration for <name>.
CLIPS does not recognize <name> as a declared function and gives this error message.

Example:
```
CLIPS> (xyz)
```

[EXPRNPSR4] $ Sequence operator not a valid argument for <name>.
The sequence expansion operator cannot be used with certain functions.

Example:
```
CLIPS> (set-sequence-operator-recognition TRUE)
FALSE
CLIPS> (defrule foo (x $?y) => (assert (x1 $?y)))
```

[FACTMCH1] This error occurred in the fact pattern network
Currently active fact: <newly assert fact>
Problem resides in slot <slot name>
Of pattern #<pattern-number> in rule <rule name>
This error pinpoints other evaluation errors associated with evaluating an expression within the pattern network. The specific pattern and field of the problem rules are identified.

[FACTMNGR1] Facts may not be retracted during pattern-matching
or
[FACTMNGR2] Facts may not be asserted during pattern-matching
Functions used on the LHS of a rule should not have side effects (such as the creation of a new instance or fact).

Example:
```
CLIPS>
(defrule error
  (test (assert (blah)))
=>)
CLIPS> (reset)
```

[FACTRHS1] Implied deftemplate <name> cannot be created with binary load in effect.
This error occurs when an assert is attempted for a deftemplate which does not exist in a runtime or active **bload** image. In other situations, CLIPS will create an implied deftemplate if one does not already exist.

Example:
```
CLIPS> (clear)
CLIPS> (bsave error.bin)
TRUE
CLIPS> (bload error.bin)
TRUE
CLIPS> (assert (error))
```

[GENRCCOM1] No such generic function <name> in function undefmethod.
This error occurs when the generic function name passed to the undefmethod function does not exist.

Example:
```
CLIPS> (clear)
CLIPS> (undefmethod yak 3)
```

[GENRCCOM2] Expected a valid method index in function undefmethod.
This error occurs when an invalid method index is passed to undefmethod (e.g. a negative integer or a symbol other than *).

Example:
```
CLIPS> (defmethod foo ())
CLIPS> (undefmethod foo a))
```

[GENRCCOM3] Incomplete method specification for deletion.
It is illegal to specify a non-wildcard method index when a wildcard is given for the generic function in the function **undefmethod**.

Example:
```
CLIPS> (undefmethod * 1)
```

[GENRCCOM4] Cannot remove implicit system function method for generic function <name>.
A method corresponding to a system defined function cannot be deleted.

Example:
```
CLIPS> (defmethod integer ((?x SYMBOL)) 0)
CLIPS> (list-defmethods integer)
integer #SYS1  (NUMBER)
integer #2  (SYMBOL)
For a total of 2 methods.
CLIPS> (undefmethod integer 1)
```

[GENRCEXE1] No applicable methods for <name>.
The generic function call arguments do not satisfy any method's parameter restrictions.

Example:
```
CLIPS> (defmethod foo ())
CLIPS> (foo 1 2)
```

[GENRCEXE2] Shadowed methods not applicable in current context.
No shadowed method is available when the function **call-next-method** is called.

Example:
```
CLIPS> (call-next-method)
```

[GENRCEXE3] Unable to determine class of <value> in generic function <name>.
The class or type of a generic function argument could not be determined for comparison to a method type restriction.

Example:
```
CLIPS> (defmethod foo ((?a INTEGER)))
CLIPS> (foo [bogus-instance])
```

[GENRCEXE4] Generic function <name> method #<index> is not applicable to the given arguments.
This error occurs when **call-specific-method** is called with an inappropriate set of arguments for the specified method.

Example:
```
CLIPS> (defmethod foo ())
CLIPS> (call-specific-method foo 1 abc)
```

[GENRCFUN1] Defgeneric <name> cannot be modified while one of its methods is executing.
Defgenerics can't be redefined while one of their methods is currently executing.

Example:
```
CLIPS> (defgeneric foo)
CLIPS> (defmethod foo () (build "(defgeneric foo)"))
CLIPS> (foo)
```

[GENRCFUN2] Unable to find method <name> #<index> in function <name>.
No generic function method of the specified index could be found by the named function.

Example:
```
CLIPS> (defmethod foo 1 ())
CLIPS> (ppdefmethod foo 2)
```

[GENRCFUN3] Unable to find generic function <name> in function <name>.
No generic function method of the specified index could be found by the named function.

Example:
```
CLIPS> (preview-generic balh)
```

[GENRCPSR1] Expected ')' to complete defgeneric.
A right parenthesis completes the definition of a generic function header.

Example:
```
CLIPS> (defgeneric foo ())
```

[GENRCPSR2] New method #<index1> would be indistinguishable from method #<index2>.
An explicit index has been specified for a new method that does not match that of an older method which has identical parameter restrictions.

Example:
```
CLIPS> (defmethod foo 1 ((?a INTEGER)))
CLIPS> (defmethod foo 2 ((?a INTEGER)))
```

[GENRCPSR3] Defgenerics are not allowed to replace constructs.
A generic function cannot have the same name as any construct.

[GENRCPSR4] Deffunction <name> imported from module <module name> conflicts with this defgeneric.
A deffunction cannot have the same name as any generic function imported from another module.

Example:
```
CLIPS> (defmodule MAIN (export ?ALL))
CLIPS> (deffunction foo ())
CLIPS> (defmodule FOO (import MAIN ?ALL))
CLIPS> (defmethod foo)
```

[GENRCPSR5] Defgenerics are not allowed to replace deffunctions.
A generic function cannot have the same name as any deffunction.

[GENRCPSR6] Method index out of range.
A method index cannot be greater than the maximum value of an integer or less than 1.

Example:
```
CLIPS> (defmethod foo 0 ())
```
[GENRCPSR7] Expected a '(' to begin method parameter restrictions.
A left parenthesis must begin a parameter restriction list for a method.

Example:
```
CLIPS> (defmethod foo)
```

[GENRCPSR8] Expected a variable for parameter specification.
A method parameter with restrictions must be a variable.

Example:
```
CLIPS> (defmethod foo ((abc)))
```

[GENRCPSR9] Expected a variable or '(' for parameter specification.
A method parameter must be a variable with or without restrictions.

Example:
```
CLIPS> (defmethod foo (abc))
```

[GENRCPSR10] Query must be last in parameter restriction.
A query parameter restriction must follow a type parameter restriction (if any).

Example:
```
CLIPS> (defmethod foo ((?a (< ?a 1) INTEGER)))
```

[GENRCPSR11] Duplicate classes/types not allowed in parameter restriction.
A method type parameter restriction may have only a single occurrence of a particular class.

Example:
```
CLIPS> (defmethod foo ((?a INTEGER INTEGER)))
```

[GENRCPSR12] Binds are not allowed in query expressions.
Binding new variables in a method query parameter restriction is illegal.

Example:
```
CLIPS> (defmethod foo ((?a (bind ?b 1))))
```

[GENRCPSR13] Expected a valid class/type name or query.
Method parameter restrictions consist of zero or more class names and an optional query expression.

Example:
```
CLIPS> (defmethod foo ((?a 34)))
```

[GENRCPSR14] Unknown class/type in method.
Classes in method type parameter restrictions must already be defined.

Example:
```
CLIPS> (defmethod foo ((?a bogus-class)))
```

[GENRCPSR15] Class <name> is redundant.
All classes in a method type parameter restriction should be unrelated.

Example:
```
CLIPS> (defmethod foo ((?a INTEGER NUMBER)))
```

[GENRCPSR16] The system function <name> cannot be overloaded.
Some system functions canot be overloaded.

Example:
```
CLIPS> (defmethod if ())
```

[GENRCPSR17] Cannot replace the implicit system method #<integer>.
A system function can not be overloaded with a method that has the exact number and types of arguments.

Example:
```
CLIPS> (defmethod integer ((?x NUMBER)) (* 2 ?x))
```

[GLOBLDEF1] Global variable <variable name> is unbound.
A global variable must be defined before it can be accessed at the command prompt or elsewhere.

Example:
```
CLIPS> (clear)
CLIPS> ?*x*
```

[GLOBLPSR1] Global variable <variable name> was referenced, but is not defined.
A global variable must be defined before it can be accessed at the command prompt or elsewhere.

Example:
```
CLIPS> (clear)
CLIPS> ?*x*
```

[INHERPSR1] A class may not have itself as a superclass.
A class may not inherit from itself.

Example:
```
CLIPS> (defclass A (is-a A))
```

[INHERPSR2] A class may inherit from a superclass only once.
All direct superclasses of a class must be unique.

Example:
```
CLIPS> (defclass A (is-a USER USER))
```
[INHERPSR3] A class must be defined after all its superclasses.
Subclasses must be defined last.

Example:
```
CLIPS> (defclass B (is-a A))
```

[INHERPSR4] A class must have at least one superclass.
All user-defined classes must have at least one direct superclass.

Example:
```
CLIPS> (defclass A (is-a))
```

[INHERPSR5] Partial precedence list formed: <classa> <classb> … <classc>
Precedence loop in superclasses: <class1> <class2> … <classn> <class1>
No class precedence list satisfies the rules specified in section 9.3.1.1 for the given direct superclass list. The message shows a conflict for <class1> because the precedence implies that <class1> must both precede and succeed <class2> through <classn>. The full loop can be used to help identify which particular classes are causing the problem. This loop is not necessarily the only loop in the precedence list; it is the first one detected. The part of the precedence list which was successfully formed is also listed.

Example:
```
CLIPS> (defclass A (is-a MULTIFIELD FLOAT SYMBOL))
CLIPS> (defclass B (is-a SYMBOL FLOAT))
CLIPS> (defclass C (is-a A B))
```

[INHERPSR6] A user-defined class cannot be a subclass of <name>.
The INSTANCE, INSTANCE-NAME, and INSTANCE-ADDRESS classes cannot have any subclasses.

Example:
```
CLIPS> (defclass A (is-a INSTANCE))
```

[INSCOM1] Undefined type in function <name>.
The evaluation of an expression yielded something other than a recognized class or primitive type.

[INSFILE1] Function <function-name> could not completely process file <name>.
This error occurs when an instance definition is improperly formed in the input file for the **load-instances**, **restore-instances**, or **bload-instances** command.

Example:
```
CLIPS> (load-instances bogus.txt)
```
[INSFILE2] File <file-name> is not a binary instances file.
or
[INSFILE3] File <file-name> is not a compatible binary instances file.
This error occurs when bload-instances attempts to load a file that was not created with bsave-instances or when the file being loaded was created by a different version of CLIPS.

Example:
```
CLIPS> (reset)
CLIPS> (save-instances foo.ins)
1
CLIPS> (bload-instances foo.ins)
```

[INSFILE4] Function 'bload-instances' is unable to load instance <instance-name>.
This error occurs when an instance specification in the input file for the **bload-instances** command could not be created.

Example:

```
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS> (make-instance of A)
[gen1]
CLIPS> (bsave-instances foo.bin)
1
CLIPS> (clear)
CLIPS> (defclass A (is-a USER))
CLIPS> (bload-instances foo.bin)
```

[INSFUN1] Expected a valid instance in function <name>.
The named function expected an instance-name or address as an argument.

Example:
```
CLIPS> (initialize-instance 34)
```

[INSFUN2] No such instance <name> in function <name>.
This error occurs when the named function cannot find the specified instance.

Example:
```
CLIPS> (instance-address [bogus-instance])
```

[INSFUN3] No such slot <name> in function <name>.
This error occurs when the named function cannot find the specified slot in an instance or class.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (slot-writablep A b)
```

[INSFUN4] Invalid instance-address in function <name>, argument #<integer>.
This error occurs when an attempt is made to use the address of a deleted instance.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (make-instance a of A)
[a]
CLIPS> (defglobal ?*x* = (instance-address a))
CLIPS> (make-instance a of A)
[a]
CLIPS> (class ?*x*)
```

[INSFUN5] Cannot modify reactive instance slots while pattern-matching is in process.
CLIPS does not allow reactive instance slots to be changed while pattern-matching is taking place. Functions used on the LHS of a rule should not have side effects (such as the changing slot values).

Example:
```
CLIPS>
(defclass FOO (is-a USER)
              (role concrete)
              (pattern-match reactive)
              (slot x (create-accessor read-write)))
CLIPS> (make-instance x of FOO)
```

```
[x]
CLIPS> (defrule BAR (x) (test (send [x] put-x 3)) =>)
CLIPS> (assert (x))
```

[INSFUN6] Unable to pattern-match on shared slot <name> in class <name>.
This error occurs when the number of simultaneous class hierarchy traversals is exceeded while pattern-matching on a shared slot. See the related error message [CLASSFUN2] for more details.

[INSFUN7] The value<multifield-value> is illegal for single-field slot <name> of instance <name> found in <function-call or message-handler>.
Single-field slots in an instance can hold only one atomic value.

Example:

```
CLIPS> (set-static-constraint-checking FALSE)
TRUE
CLIPS>
(defclass FOO (is-a USER)
  (role concrete)
  (slot foo))
CLIPS>
(defmessage-handler FOO error ()
  (bind ?self:foo 1 2 3))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (send [foo] error)
```

[INSFUN8] Void function illegal value for slot <name> of instance <name> found in <function-call or message-handler>.
Only functions which have a return value can be used to generate values for an instance slot.

Example:

```
CLIPS> (set-static-constraint-checking FALSE)
TRUE
CLIPS>
(defclass FOO (is-a USER)
  (role concrete)
  (slot foo))
CLIPS>
(defmessage-handler FOO error ()
  (bind ?self:foo (instances)))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (send [foo] error)
```

[INSMNGR1] Expected a valid name for new instance.
make-instance expects a symbol or an instance-name for the name of a new instance.

Example:

```
CLIPS> (make-instance 34 of A)
```

[INSMNGR2] Expected a valid class name for new instance.
make-instance expects a symbol for the class of a new instance.

Example:
```
CLIPS> (make-instance a of 34)
```

[INSMNGR3] Cannot create instances of abstract class <name>.
Direct instances of abstract classes, such as the predefined system classes, are illegal.

Example:
```
CLIPS> (make-instance [foo] of USER)
```

[INSMGNR4] The instance <name> has a slot-value which depends on the instance definition.
The initialization of an instance is recursive in that a slot-override or default-value tries to create or reinitialize the same instance.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo))
CLIPS>
(make-instance a of A (foo (make-instance a of A)))
```

[INSMNGR5] Unable to delete old instance <name>.
make-instance will attempt to delete an old instance of the same name if it exists. This error occurs if that deletion fails.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS>
(defmessage-handler A delete around ()
  (if (neq (instance-name ?self) [a]) then
    (call-next-handler)))
CLIPS> (make-instance a of A)
CLIPS> (make-instance a of A)
```

[INSMNGR6] Cannot delete instance <name> during initialization.
The evaluation of a slot-override in **make-instance** or **initialize-instance** attempted to delete the instance.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo))
CLIPS>
(defmessage-handler A put-foo after ($?any)
  (delete-instance))
CLIPS> (make-instance a of A (foo 2))
```

[INSMNGR7] Instance <name> is already being initialized.
An instance cannot be reinitialized during initialization.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (make-instance a of A)
CLIPS>
(defmessage-handler A init after ()
   (initialize-instance ?self))
CLIPS> (initialize-instance a)
CLIPS> (send [a] try)
```

[INSMNGR8] An error occurred during the initialization of instance <name>.
This message is displayed when an evaluation error occurs while the **init** message is executing for an instance.

[INSMNGR9] Expected a valid slot name for slot-override.
make-instance and **initialize-instance** expect symbols for slot names.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (make-instance a of A (34 override-value))
```

[INSMNGR10] Cannot create instances of reactive classes while pattern-matching is in process.
CLIPS does not allow instances of reactive classes to be created while pattern-matching is taking place. Functions used on the LHS of a rule should not have side effects (such as the creation of a new instance or fact).

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete) (pattern-match reactive))
CLIPS> (defrule BAR (x) (test (make-instance of FOO)) =>)
CLIPS> (assert (x))
```

[INSMNGR11] Invalid module specifier in new instance name.
This error occurs when the module specifier in the instance-name is illegal (such as an undefined module name).

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete))
CLIPS> (make-instance BOGUS::x of FOO)
```

[INSMNGR12] Cannot delete instances of reactive classes while pattern-matching is in process.
CLIPS does not allow instances of reactive classes to be deleted while pattern-matching is taking place. Functions used on the LHS of a rule should not have side effects (such as the deletion of a new instance or the retraction of a fact).

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete) (pattern-match reactive))
CLIPS> (make-instance x of FOO)
[x]
```

```
CLIPS> (defrule BAR (x) (test (send [x] delete)) =>)
CLIPS> (assert (x))
```

[INSMNGR13] Slot <slot-name> does not exist in instance <instance-name>.
This error occurs when the slot name of a slot override does not correspond to any of the valid slot names for an instance.

Example:

```
CLIPS> (defclass FOO (is-a USER) (role concrete))
CLIPS> (make-instance of FOO (x 3))
```

[INSMNGR14] Override required for slot <slot-name> in instance <instance-name>.
If the ?NONE keyword was specified with the default attribute for a slot, then a slot override must be provided when an instance containing that slot is created.

Example:

```
CLIPS> (defclass FOO (is-a USER)
                     (role concrete)
                     (slot x (default ?NONE)))
CLIPS> (make-instance of FOO)
```

[INSMNGR15] init-slots not valid in this context.
The special function **init-slots** (for initializing slots of an instance to the class default values) can only be called during the dispatch of an **init** message for an instance, i.e., in an **init** message-handler.

Example:

```
CLIPS> (defclass FOO (is-a USER))
CLIPS>
(defmessage-handler FOO error ()
  (init-slots))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (send [foo] error)
```

[INSMNGR16] The instance name <instance-name> is in use by an instance of class <class-name>.
An instance of one class cannot be created using an instance name belonging to a different class.

Example:

```
CLIPS> (defclass A (is-a USER))
CLIPS> (defclass B (is-a USER))
CLIPS> (make-instance [a] of A)
[a]
CLIPS> (make-instance [a] of B)
```

[INSMODDP1] Direct/message-modify message valid only in modify-instance.
The **direct-modify** and **message-modify** message-handlers attached to the class **USER** can only be called as a result of the appropriate message being sent.by the **modify-instance** or **message-modify-instance** functions. Additional handlers may be defined, but the message can only be sent in this context.

Example:
```
CLIPS> (defclass FOO (is-a USER))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (send [foo] direct-modify 0)
```

[INSMODDP2] Direct/message-duplicate message valid only in duplicate-instance.
The **direct-duplicate** and **message-duplicate** message-handlers attached to the class **USER** can only be called as a result of the appropriate message being sent.by the **duplicate-instance** or **message-duplicate-instance** functions. Additional handlers may be defined, but the message can only be sent in this context.

Example:
```
CLIPS> (defclass FOO (is-a USER))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (send [foo] direct-duplicate 0 0)
```

[INSMODDP3] Instance copy must have a different name in duplicate-instance.
If an instance-name is specified for the new instance in the call to **duplicate-instance**, it must be different from the source instance's name.

Example:
```
CLIPS> (defclass FOO (is-a USER))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (duplicate-instance foo to foo)
```

[INSMULT1] Function <name> cannot be used on single-field slot <name> in instance <name>.
The functions described in section 12.13.4.12, such as **slot-insert$**, can only operate on multifield slots.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo))
CLIPS> (make-instance a of A)
[a]
CLIPS> (slot-insert$ a foo 1 abc def)
```

[INSQYPSR1] Duplicate instance-set member variable name in function <name>.
Instance-set member variables in an instance-set query function must be unique.

Example:
```
CLIPS> (any-instancep ((?a OBJECT) (?a OBJECT)) TRUE)
```

[INSQYPSR2] Binds are not allowed in instance-set query in function <name>.
An instance-set query cannot bind variables.

Example:
```
CLIPS>
(any-instancep ((?a OBJECT) (?b OBJECT))
  (bind ?c 1))
```

[INSQYPSR3] Cannot rebind instance-set member variable <name> in function <name>.
Instance-set member variables cannot be changed within the actions of an instance-set query function.

Example:
```
CLIPS>
(do-for-all-instances ((?a USER))
  (if (slot-existp ?a age) then
     (> ?a:age 30))
  (bind ?a (send ?a get-brother)))
```

[IOFUN1] Illegal logical name used for <function name> function.
A logical name must be either a symbol, string, instance-name, float, or integer.

Example:
```
(printout (create$ a b c) x)
```

[IOFUN2] Logical name <logical name> already in use.
A logical name cannot be associated with two different files.

Example:
```
CLIPS> (open "foo.txt" foo "w")
TRUE
CLIPS> (open "foo2.txt" foo "w")
```

[MEMORY1] Out of memory
This error indicates insufficient memory exists to expand internal structures enough to allow continued operation (causing an exit to the operating system).

[MISCFUN1] The function 'expand$' must be used in the argument list of a function call.
or
[MISCFUN1] Sequence expansion must be used in the argument list of a function call.
Sequence expansion and the expand$ function may not be used unless it is within the argument list of another function.

Example:
```
CLIPS> (expand$ (create$ a b c))
```

[MODULDEF1] Illegal use of the module specifier.
The module specifier can only be used as part of a defined construct's name or as an argument to a function.

Example:
```
CLIPS> (deffunction y ())
CLIPS> (MAIN::y)
```

[MODULPSR1] Module <module name> does not export any constructs.
or
[MODULPSR1] Module <module name> does not export any <construct type> constructs.
or
[MODULPSR1] Module <module name> does not export the <construct type> <construct name>.
A construct cannot be imported from a module unless the defmodule exports that construct.

Example:
```
CLIPS> (clear)
CLIPS> (defmodule BAR)
CLIPS> (deftemplate BAR::bar)
CLIPS> (defmodule FOO (import BAR deftemplate bar)))
```

[MSGCOM1] Incomplete message-handler specification for deletion.
It is illegal to specify a non-wildcard handler index when a wildcard is given for the class in the external C function **UndefmessageHandler()**. This error can only be generated when a user-defined external function linked with CLIPS calls this function incorrectly.

[MSGCOM2] Unable to find message-handler <name> <type> for class <name> in function <name>.
This error occurs when the named function cannot find the specified message-handler.

Example:
```
CLIPS> (ppdefmessage-handler USER foo around)
```

[MSGCOM3] Unable to delete message-handlers.
This error occurs when a message-handler can't be deleted (such as when a binary image is loaded).

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete))
CLIPS> (defmessage-handler FOO bar ())
CLIPS> (bsave foo.bin)
TRUE
CLIPS> (bload foo.bin)
TRUE
CLIPS> (undefmessage-handler FOO bar)
```

[MSGFUN1] No applicable primary message-handlers found for <message>.
No primary message-handler attached to the object's classes matched the name of the message.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] bogus-message)
```

[MSGFUN2] Message-handler <name> <type> in class <name> expected exactly/at least <number> argument(s).
The number of message arguments was inappropriate for one of the applicable message-handlers.

Example:
```
CLIPS> (defclass A (is-a USER))
CLIPS> (defmessage-handler USER foo (?a ?b))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] foo)
```

[MSGFUN3] Write access denied for slot <name> in instance <name>.
This error occurs when an attempt is made to change the value of a read-only slot.

Example:
```
CLIPS>
(defclass A (is-a USER)
  (slot foo (default 100)
            (read-only)))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] put-foo)
```

[MSGFUN4] The function <function> may only be called from within message-handlers.
The named function operates on the active instance of a message and thus can only be called by message-handlers.

Example:
```
CLIPS> (ppinstance)
```

[MSGFUN5] The function <function> operates only on instances.
The named function operates on the active instance of a message and can only handle instances of user-defined classes (not primitive type objects).

Example:
```
CLIPS>
(defmessage-handler INTEGER print ()
  (ppinstance))
CLIPS> (send 34 print)
```

[MSGFUN6] Private slot <slot-name> of class <class-name> cannot be accessed directly by handlers attached to class <class-name>
A subclass which inherits private slots from a superclass may not access those slots using the ?self variable. This error can also occur when a superclass tries to access via **dynamic-put** or **dynamic-get** a private slot in a subclass.

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete) (slot x))
CLIPS> (defclass BAR (is-a FOO))
CLIPS> (defmessage-handler BAR yak () ?self:x)
```

[MSGFUN7] Unrecognized message-handler type in defmessage-handler in function <function>.
Allowed message-handler types include primary, before, after, and around.

Example:
```
CLIPS> (defmessage-handler USER foo behind ())
```

[MSGFUN8] Unable to delete message-handler(s) from class <name>.
This error occurs when an attempt is made to delete a message-handler attached to a class for which any of the message-handlers are executing.

Example:
```
CLIPS> (defclass FOO (is-a USER))
CLIPS>
(defmessage-handler FOO error ()
   (undefmessage-handler FOO error primary))
CLIPS> (make-instance foo of FOO)
[foo]
CLIPS> (send [foo] error)
```

[MSGPASS1] Shadowed message-handlers not applicable in current context.
No shadowed message-handler is available when the function **call-next-handler** or **override-next-handler** is called.

Example:
```
CLIPS> (call-next-handler)
```

[MSGPASS2] No such instance <name> in function <name>.
This error occurs when the named function cannot find the specified instance.

Example:
```
CLIPS> (instance-address [bogus-instance])
```

[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 ())
```

[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 ())
```

[MSGPSR4] Illegal slot reference in parameter list.
Direct slot references are allowed only within message-handler bodies.

Example:

```
CLIPS> (defmessage-handler USER foo (?self:bar))
```

[MSGPSR5] Active instance parameter cannot be changed.
?self is a reserved parameter for the active instance.

Example:
```
CLIPS>
(defmessage-handler USER foo ()
  (bind ?self 1))
```

[MSGPSR6] No such slot <name> in class <name> for ?self reference.
The symbol following the ?self: reference must be a valid slot for the class.

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete) (slot x))
CLIPS> (defmessage-handler FOO bar () ?self:y)
```

[MSGPSR7] Illegal value for ?self reference.
The symbol following the ?self: reference must be a symbol.

Example:
```
CLIPS> (defclass FOO (is-a USER) (role concrete) (slot x))
CLIPS> (defmessage-handler FOO bar () ?self:7)
```

[MSGPSR8] Message-handlers cannot be attached to the class <name>.
Message-handlers cannot be attached to the INSTANCE, INSTANCE-ADDRESS, or INSTANCE-NAME classes.

Example:
```
CLIPS> (defmessage-handler INSTANCE foo ())
```

[MULTIFUN1] Multifield index <index> out of range 1..<end range> in function <name>
or
[MULTIFUN1] Multifield index range <start>...<end> out of range 1..<end range> in function <name>
This error occurs when a multifield manipulation function is passed a single index or range of indices that does not fall within the specified range of allowed indices.

Example:
```
CLIPS> (delete$ (create$ a b c) 4 4)
```

[MULTIFUN2] Cannot rebind field variable in function <function>.
The field variable (if specified) cannot be rebound within the body of the progn$ or foreach function.

Example:
```
CLIPS> (progn$ (?field (create$ a)) (bind ?field 3))
```

[OBJRTBLD1] No objects of existing classes can satisfy pattern.
No objects of existing classes could possibly satisfy the pattern. This error usually occurs when a restriction placed on the is-a attribute is incompatible with slot restrictions before it in the pattern.

Example:
```
CLIPS> (defclass A (is-a USER) (slot foo))
CLIPS> (defrule error (object (foo ?) (is-a ~A)) =>)
```

[OBJRTBLD2] No objects of existing classes can satisfy <attribute-name> restriction in object pattern.
The restrictions on <attribute> are such that no objects of existing classes (which also satisfy preceding restrictions) could possibly satisfy the pattern.

Example:
```
CLIPS> (defrule error (object (bad-slot ?)) =>)
```
[OBJRTBLD3] No objects of existing classes can satisfy pattern #<pattern-num>.
No objects of existing classes could possibly satisfy the pattern. This error occurs when the constraints for a slot as given in the defclass(es) are incompatible with the constraints imposed by the pattern.

Example:
```
CLIPS>
(defclass FOO (is-a USER)
  (slot bar (type INTEGER)))
CLIPS>
(defclass BAR (is-a USER)
  (slot bar (type SYMBOL))
  (slot woz))
CLIPS>
(defrule error
  (x abc)
  (object (bar 100) (woz ?))
  (y def)
=>)
```

[OBJRTBLD4] Multiple restrictions on attribute <attribute-name> not allowed.
Only one restriction per attribute is allowed per object pattern.

Example:
```
CLIPS> (defrule error (object (is-a ?) (is-a ?)) =>)
```

[OBJRTBLD5] Undefined class in object pattern.
Object patterns are applicable only to classes of objects which are already defined.

Example:
```
CLIPS> (defrule error (object (is-a BOGUS)) =>)
```

[OBJRTMCH1] This error occurred in the object pattern network
Currently active instance: <instance-name>
Problem resides in slot <slot name> field #<field-index>
Of pattern #<pattern-number> in rule(s):
<problem-rules>+
This error pinpoints other evaluation errors associated with evaluating an expression within the object pattern network. The specific pattern and field of the problem rules are identified.

[PATTERN1] The symbol <symbol name> has special meaning and may not be used as a <use name>.
Certain keywords have special meaning to CLIPS and may not be used in situations that would cause an ambiguity.

Example:
```
CLIPS> (deftemplate exists (slot x))
```

[PATTERN2] Single and multifield constraints cannot be mixed in a field constraint
Single and multifield variable constraints cannot be mixed in a field constraint (this restriction does not include variables passed to functions with the predicate or return value constraints).

Example:
```
CLIPS> (defrule foo (a ?x $?y ?x&~$?y) =>)
```

[PRCCODE1] Attempted to call a <construct> which does not exist.
In a CLIPS configuration without deffunctions and/or generic functions, an attempt was made to call a deffunction or generic function from a binary image generated by the **bsave** command.

[PRCCODE2] Functions without a return value are illegal as <construct> arguments.
An evaluation error occurred while examining the arguments for a deffunction, generic function or message.

Example:
```
CLIPS> (defmethod foo (?a))
CLIPS> (foo (instances))
```

[PRCCODE3] Undefined variable <name> referenced in <where>.
Local variables in the actions of a deffunction, method, message-handler, or defrule must reference parameters, variables bound within the actions with the **bind** function, or variables bound on the LHS of a rule.

Example:
```
CLIPS> (defrule foo => (+ ?a 3))
```

[PRCCODE4] Execution halted during the actions of <construct> <name>.
This error occurs when the actions of a rule, deffunction, generic function method or message-handler are prematurely aborted due to an error.

[PRCCODE5] Variable <name> unbound [in <construct> <name>].
This error occurs when local variables in the actions of a deffunction, method, message-handler, or defrule becomes unbound during execution as a result of calling the **bind** function with no arguments.

Example:
```
CLIPS> (deffunction foo () (bind ?a) ?a)
CLIPS> (foo)
```

[PRCCODE6] This error occurred while evaluating arguments for the <construct> <name>.
An evaluation error occurred while examining the arguments for a deffunction, generic function method or message-handler.

Example:
```
CLIPS> (deffunction foo (?a))
CLIPS> (foo (+ (eval "(gensym)") 2))
```

[PRCCODE7] Duplicate parameter names not allowed.
Deffunction, method or message-handler parameter names must be unique.

Example:
```
CLIPS> (defmethod foo ((?x INTEGER) (?x FLOAT)))
```

[PRCCODE8] No parameters allowed after wildcard parameter.
A wildcard parameter for a deffunction, method or message-handler must be the last parameter.

Example:
```
CLIPS> (defmethod foo (($?x INTEGER) (?y SYMBOL)))
```

[PRCDRPSR1] Cannot rebind count variable in function loop-for-count.
The special variable ?count cannot be rebound within the body of the loop-for-count function.

Example:
```
CLIPS> (loop-for-count (?count 10) (bind ?count 3))
```

[PRCDRPSR2] The return function is not valid in this context.
or
[PRCDRPSR2] The break function is not valid in this context.
The return and break functions can only be used within certain contexts (e.g. the break function can only be used within a while loop and certain instance set query functions).

Example:
```
CLIPS> (return 3)
```

[PRCDRPSR3] Duplicate case found in switch function.
A case may be specified only once in a switch statement.

Example:
```
CLIPS> (switch a (case a then 8) (case a then 9))
```

[PRNTUTIL1] Unable to find <item> <item-name>
This error occurs when CLIPS cannot find the named item (check for typos).

[PRNTUTIL2] Syntax Error: Check appropriate syntax for <item>
This error occurs when the appropriate syntax is not used.

Example:
```
CLIPS> (if (> 3 4))
```

[PRNTUTIL3]
***** CLIPS SYSTEM ERROR *****
ID = <error-id>
CLIPS data structures are in an inconsistent or corrupted state.
This error may have occurred from errors in user defined code.
****
This error indicates an internal problem within CLIPS (which may have been caused by user defined functions or other user code). If the problem cannot be located within user defined code, then the <error-id> should be reported.

[PRNTUTIL4] Unable to delete <item> <item-name>
This error occurs when CLIPS cannot delete the named item (e.g. a construct might be in use). One example which will cause this error is an attempt to delete a deffunction or generic function which is used in another construct (such as the RHS of a defrule or a default-dynamic facet of a defclass slot).

[PRNTUTIL5] The <item> has already been parsed.
This error occurs when CLIPS has already parsed an attribute or declaration.

[PRNTUTIL6] Local variables cannot be accessed by <function or construct>.
This error occurs when a local variable is used by a function or construct that cannot use global variables.

Example:
```
CLIPS> (deffacts info (fact ?x))
```

[PRNTUTIL7] Attempt to divide by zero in <function-name> function.
This error occurs when a function attempts to divide by zero.

Example:
```
CLIPS> (/ 3 0)
```

[PRNTUTIL8] This error occurred while evaluating the salience [for rule <name>]
When an error results from evaluating a salience value for a rule, this error message is given.

[PRNTUTIL9] Salience value out of range <min> to <max>
The range of allowed salience has an explicit limit; this error message will result if the value is out of that range.

Example:
```
CLIPS> (defrule error (declare (salience 20000)) =>)
```

[PRNUTIL10] Salience value must be an integer value.
Salience requires a integer argument and will otherwise result in this error message.

Example:
```
CLIPS> (defrule error (declare (salience a)) =>)
```

[PRNUTIL11] The fact <fact-id> has been retracted.
This error occurs when a function expecting a fact address argument is provided a retracted fact.

Example:
```
CLIPS> (bind ?f (assert (a b c)))
<Fact-1>
CLIPS> (retract ?f)
CLIPS> (retract ?f)
```

[PRNUTIL12] The variable/slot reference ?<variable>:<slot> cannot be resolved because the referenced fact <fact-id> has been retracted.
This error occurs when using shorthand slot notation with a retracted fact.

Example:
```
CLIPS> (deftemplate point (slot x) (slot y))
CLIPS> (assert (point (x 1) (y 2)))
<Fact-1>
CLIPS> (do-for-fact ((?p point)) TRUE (retract ?p) (+ ?p:x ?p:y))
```

[PRNUTIL13] The variable/slot reference ?<variable>:<slot> is invalid because the referenced fact <fact-id> does not contain the specified slot.
This error occurs when using shorthand slot notation for a fact that does not contain the specified slot.

Example:
```
CLIPS> (deftemplate point (slot x) (slot y))
CLIPS> (assert (point (x 1) (y 2)))
<Fact-1>
CLIPS> (do-for-fact ((?p point)) TRUE (+ ?p:x ?p:z))
```

[PRNUTIL14] The variable/slot reference ?<variable>:<slot> is invalid because slot names must be symbols.
This error occurs when using shorthand slot notation with a non-symbolic slot name.

Example:
```
CLIPS> (clear)
CLIPS> (deftemplate point (slot x) (slot y))
CLIPS> (do-for-fact ((?p point)) TRUE (+ ?p:x ?p:37))
```

[PRNUTIL15] The variable/slot reference ?<variable>:<slot> cannot be resolved because the referenced instance <instance-name> has been deleted.
This error occurs when using shorthand slot notation with a deleted instance.

Example:
```
CLIPS> (defclass POINT (is-a USER) (slot x) (slot y))
CLIPS> (make-instance p1 of POINT (x 1) (y 2))
[p1]
CLIPS> (do-for-all-instances ((?p POINT)) TRUE (send ?p delete) (+ ?p:x ?p:y))
```

[PRNUTIL16] The variable/slot reference ?<variable>:<slot> is invalid because the referenced instance <instance-name> does not contain the specified slot.
This error occurs when using shorthand slot notation for an instance that does not contain the specified slot.

Example:
```
CLIPS> (defclass POINT (is-a USER) (slot x) (slot y))
CLIPS> (make-instance p1 of POINT (x 1) (y 2))
[p1]
CLIPS> (do-for-all-instances ((?p POINT)) TRUE (+ ?p:x ?p:z))
```

[ROUTER1] Logical name <logical_name> was not recognized by any routers
This error results because "Hello" is not recognized as a valid router name.

Example:
```
CLIPS> (printout "Hello" crlf)
```

[RULECSTR1] Variable <variable name> in CE #<integer> slot <slot name> has constraint conflicts which make the pattern unmatchable.
or
[RULECSTR1] Variable <variable name> in CE #<integer> field #<integer> has constraint conflicts which make the pattern unmatchable.
or
[RULECSTR1] CE #<integer> slot <slot name> has constraint conflicts which make the pattern unmatchable.
or

[RULECSTR1] CE #<integer> field #<integer> has constraint conflicts which make the pattern unmatchable.
This error occurs when slot value constraints (such as allowed types) prevents any value from matching the slot constraint for a pattern.

Example:
```
CLIPS> (deftemplate foo (slot x (type SYMBOL)))
CLIPS> (deftemplate bar (slot x (type FLOAT)))
CLIPS> (defrule yak (foo (x ?x)) (bar (x ?x)) =>)
```

**[RULECSTR2] Previous variable bindings of <variable name> caused the type restrictions for argument #<integer> of the expression <expression>
found in CE#<integer> slot <slot name> to be violated.**
This error occurs when previous variable bindings and constraints prevent a variable from containing a value which satisfies the type constraints for one of a function's parameters.

Example:
```
CLIPS> (deftemplate foo (slot x (type SYMBOL)))
CLIPS> (defrule bar (foo (x ?x&:(> ?x 3))) =>)
```

**[RULECSTR3] Previous variable bindings of <variable name> caused the type restrictions for argument #<integer> of the expression <expression>
found in the rule's RHS to be violated.**
This error occurs when previous variable bindings and constraints prevent a variable from containing a value which satisfies the type constraints for one of a function's parameters.

Example:
```
CLIPS> (deftemplate foo (slot x (type SYMBOL)))
CLIPS> (defrule bar (foo (x ?x)) => (println (+ ?x 1)))
```

[RULELHS1] The logical CE cannot be used with a not/exists/forall CE.
Logical CEs can be placed outside, but not inside, a not/exists/forall CE.

Example:
```
CLIPS> (defrule error (not (logical (x))) =>)
```

[RULELHS2] A pattern CE cannot be bound to a pattern-address within a not CE
This is an illegal operation and results in an error message.

Example:
```
CLIPS> (defrule error (not ?f <- (fact)) =>)
```

[RULEPSR1] Logical CEs must be placed first in a rule
If logical CEs are used, then the first CE must be a logical CE.

Example:
```
CLIPS> (defrule error (a) (logical (b)) =>)
```

[RULEPSR2] Gaps may not exist between logical CEs
Logical CEs found within a rule must be contiguous.

Example:
```
CLIPS> (defrule error (logical (a)) (b) (logical (c)) =>)
```

[STRNGFUN1] Function build does not work in run time modules.
The build function does not work in run time modules because the code required for parsing is not available.

[SYSDEP1] No file found for <option> option.
This message occurs if the –f, -f2, or -l option is used when executing CLIPS, but no arguments are provided.

Example:
```
clips -f
```

[SYSDEP2] Invalid option <option>.
This message occurs if an invalid option is used.

Example:
```
clips -f3
```

[TEXTPRO1] Could not open file <file-name>.
This error occurs when the external text-processing system command **fetch** encounters an error when loading a file.

Example:
```
CLIPS> (fetch "bogus.txt")
```

[TEXTPRO2] File <file-name> already loaded.
This error occurs when the external text-processing system command **fetch** encounters an error when loading a file.

Example:
```
CLIPS> (fetch "file.txt")
CLIPS> (fetch "file.txt")
```

[TEXTPRO3] No entries found.
or
[TEXTPRO4] Line <number> : Previous entry not closed.
or
[TEXTPRO5] Line <number> : Invalid delimeter string.
or

[TEXTPRO6] Line <number> : Invalid entry type.
or
[TEXTPRO7] Line <number> : Non-menu entries cannot have subtopics.
or
[TEXTPRO8] Line <number> : Unmatched end marker.
These errors occurs when a file is fetched with invalid entries.

[TMPLTDEF1] Invalid slot <slot name> not defined in corresponding deftemplate <deftemplate name>
The slot name supplied does not correspond to a slot name defined in the corresponding deftemplate

Example:
```
CLIPS> (deftemplate example (slot x))
CLIPS> (defrule error (example (z 3)) =>)
```

[TMPLTDEF2] The single field slot <slot name> can only contain a single field value.
If a slot definition is specified in a template pattern or fact, the contents of the slot must be capable of matching against or evaluating to a single value.

Example:
```
CLIPS> (deftemplate example (slot x))
CLIPS> (assert (example (x)))
```

[TMPLTFUN1] Attempted to assert a multifield value into the single field slot <slot name> of deftemplate <deftemplate name>.
A multifield value cannot be stored in a single field slot.

Example:
```
CLIPS> (deftemplate foo (slot x))
CLIPS>
(defrule foo
   =>
   (bind ?x (create$ a b))
   (assert (foo (x ?x))))
CLIPS> (reset)
CLIPS> (run)
```

[TMPLTRHS1] Slot <slot name> requires a value because of its (default ?NONE) attribute.
The (default ?NONE) attribute requires that a slot value be supplied whenever a new fact is created.

Example:
```
CLIPS> (deftemplate foo (slot x (default ?NONE)))
CLIPS> (assert (foo))
```

Appendix G:
CLIPS BNF

Data Types

```
<symbol>          ::= A valid symbol as specified
                      in section 2.3.1

<string>          ::= A valid string as specified
                      in section 2.3.1

<float>           ::= A valid float as specified
                      in section 2.3.1

<integer>         ::= A valid integer as specified
                      in section 2.3.1

<instance-name>   ::= A valid instance-name as specified
                      in section 2.3.1

<number>          ::= <float> | <integer>

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

<constant>        ::= <symbol> | <string> | <integer> |
                      <float> | <instance-name>

<comment>         ::= <string>

<variable-symbol> ::= A symbol beginning with an
                      alphabetic character

<function-name>   ::= Any symbol which corresponds to a
                      system or user defined function, a
                      deffunction name, or a defgeneric
                      name

<file-name>       ::= A symbol or string which is a valid
                      file name (including path
                      information) for the operating
                      system under which CLIPS is running

<slot-name>       ::= A valid deftemplate slot name

<...-name>        ::= A <symbol> where the ellipsis
                      indicate what the symbol represents.
                      For example, <rule-name> is a symbol
                      which represents the name of a rule.
```

Variables and Expressions

```
<single-field-variable> ::= ?<variable-symbol>

<multifield-variable>   ::= $?<variable-symbol>
```

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

<variable>              ::= <single-field-variable> |
                            <multifield-variable> |
                            <global-variable>

<function-call>         ::= (<function-name> <expression>*)

<expression>            ::= <constant> | <variable> |
                            <function-call>

<action>                ::= <expression>

<...-expression>        ::= An <expression> which returns
                            the type indicated by the
                            ellipsis. For example,
                            <integer-expression> should
                            return an integer.
```

Constructs

```
<CLIPS-program> ::= <construct>*

<construct>     ::= <deffacts-construct> |
                    <deftemplate-construct> |
                    <defglobal-construct> |
                    <defrule-construct> |
                    <deffunction-construct> |
                    <defgeneric-construct> |
                    <defmethod-construct> |
                    <defclass-construct> |
                    <definstance-construct> |
                    <defmessage-handler-construct> |
                    <defmodule-construct>
```

Deffacts Construct

```
<deffacts-construct> ::= (deffacts <deffacts-name> [<comment>]
                                   <RHS-pattern>*)
```

Deftemplate Construct

```
<deftemplate-construct>
                    ::= (deftemplate <deftemplate-name>
                                     [<comment>]
                                     <slot-definition>*)

<slot-definition>  ::= <single-slot-definition> |
                       <multislot-definition>

<single-slot-definition>
                    ::= (slot <slot-name> <template-attribute>*)

<multislot-definition>
                    ::= (multislot <slot-name>
                                   <template-attribute>*)

<template-attribute>
                    ::= <default-attribute> |
                        <constraint-attribute>
```

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

Fact Specification

```
<RHS-pattern>           ::= <ordered-RHS-pattern> |
                            <template-RHS-pattern>

<ordered-RHS-pattern>   ::= (<symbol> <RHS-field>+)

<template-RHS-pattern>  ::= (<deftemplate-name> <RHS-slot>*)

<RHS-slot>              ::= <single-field-RHS-slot> |
                            <multifield-RHS-slot>

<single-field-RHS-slot> ::= (<slot-name> <RHS-field>)

<multifield-RHS-slot>   ::= (<slot-name> <RHS-field>*)

<RHS-field>             ::= <variable> |
                            <constant> |
                            <function-call>
```

Defrule Construct

```
<defrule-construct>     ::= (defrule <rule-name> [<comment>]
                               [<declaration>]
                               <conditional-element>*
                               =>
                               <action>*)

<declaration>           ::= (declare <rule-property>+)

<rule-property> ::=         (salience <integer-expression>) |
                            (auto-focus <boolean-symbol>)

<boolean-symbol> ::=        TRUE | FALSE

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

<pattern-CE>            ::= <ordered-pattern-CE> |
                            <template-pattern-CE> |
                            <object-pattern-CE>

<assigned-pattern-CE>   ::= <single-field-variable> <- <pattern-CE>

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

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

<or-CE>                 ::= (or <conditional-element>+)

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

<test-CE>               ::= (test <function-call>)
```

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

<forall-CE>             ::= (forall <conditional-element>
                                    <conditional-element>+)

<ordered-pattern-CE>    ::= (<symbol> <constraint>*)

<template-pattern-CE>   ::= (<deftemplate-name> <LHS-slot>*)

<object-pattern-CE>     ::= (object <attribute-constraint>*)

<attribute-constraint>  ::= (is-a <constraint>) |
                            (name <constraint>) |
                            (<slot-name> <constraint>*)

<LHS-slot>              ::= <single-field-LHS-slot> |
                            <multifield-LHS-slot>

<single-field-LHS-slot> ::= (<slot-name> <constraint>)

<multifield-LHS-slot>   ::= (<slot-name> <constraint>*)

<constraint>            ::= ? | $? | <connected-constraint>

<connected-constraint>
             ::= <single-constraint> |
                 <single-constraint> & <connected-constraint> |
                 <single-constraint> | <connected-constraint>

<single-constraint>     ::= <term> | ~<term>

<term>                  ::= <constant> |
                            <single-field-variable> |
                            <multifield-variable> |
                            :<function-call> |
                            =<function-call>
```

Defglobal Construct

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

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

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

Deffunction Construct

```
<deffunction-construct>
           ::= (deffunction <name> [<comment>]
                  (<regular-parameter>* [<wildcard-parameter>])
                  <action>*)

<regular-parameter>  ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

Defgeneric Construct

```
<defgeneric-construct> ::= (defgeneric <name> [<comment>])
```

Defmethod Construct

```
<defmethod-construct>
      ::= (defmethod <name> [<index>] [<comment>]
             (<parameter-restriction>*
             [<wildcard-parameter-restriction>])
             <action>*)

<parameter-restriction>
          ::=  <single-field-variable> |
               (<single-field-variable> <type>* [<query>])

<wildcard-parameter-restriction>
                  ::= <multifield-variable> |
                      (<multifield-variable> <type>* [<query>])

<type>               ::= <class-name>

<query>              ::= <global-variable> | <function-call>
```

Defclass Construct

```
<defclass-construct> ::= (defclass <name> [<comment>]
                           (is-a <superclass-name>+)
                           [<role>]
                           [<pattern-match-role>]
                           <slot>*
                           <handler-documentation>*)

<role>  ::= (role concrete | abstract)

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

<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-attribute>

<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
```

Defmessage-handler Construct

```
<defmessage-handler-construct>
             ::= (defmessage-handler <class-name>
                    <message-name> [<handler-type>] [<comment>]
                    (<parameter>* [<wildcard-parameter>])
                    <action>*)

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

<parameter>          ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

Definstances Construct

```
<definstances-construct>
                ::= (definstances <definstances-name>
                       [active] [<comment>]
                       <instance-template>*)

<instance-template>   ::= (<instance-definition>)

<instance-definition> ::= <instance-name-expression> of
                             <class-name-expression>
                             <slot-override>*

<slot-override>      ::= (<slot-name-expression> <expression>*)
```

Defmodule Construct

```
<defmodule-construct> ::= (defmodule <module-name> [<comment>]
                             <port-specification>*)

<port-specification>  ::= (export <port-item>) |
                          (import <module-name> <port-item>)

<port-item>           ::= ?ALL |
                          ?NONE |
                          <port-construct> ?ALL |
                          <port-construct> ?NONE |
                          <port-construct> <construct-name>+

<port-construct>      ::= deftemplate | defclass |
                          defglobal | deffunction |
                          defgeneric
```

Constraint Attributes

```
<constraint-attribute>
                      ::= <type-attribute> |
                          <allowed-constant-attribute> |
                          <range-attribute> |
                          <cardinality-attribute>

<type-attribute>      ::= (type <type-specification>)

<type-specification>  ::= <allowed-type>+ | ?VARIABLE

<allowed-type>        ::= SYMBOL | STRING | LEXEME |
                          INTEGER | FLOAT | NUMBER |
                          INSTANCE-NAME | INSTANCE-ADDRESS |
                          INSTANCE | EXTERNAL-ADDRESS |
                          FACT-ADDRESS

<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

<range-attribute>     ::= (range <range-specification>
                                 <range-specification>)

<range-specification> ::= <number> | ?VARIABLE

<cardinality-attribute>
                   ::= (cardinality <cardinality-specification>
                                    <cardinality-specification>)

<cardinality-specification>
                      ::= <integer> | ?VARIABLE
```

Appendix H:
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
defgeneric-module
defglobal-module
definstances-module
defrule-module
deftemplate-module
deg-grad
deg-rad
delayed-do-for-all-instances
delete$
delete-instance
dependencies
dependents
describe-class

direct-mv-delete
direct-mv-insert
direct-mv-replace
div
do-for-all-instances
do-for-instance
dribble-off
dribble-on
duplicate
duplicate-instance
duplicate-instance
dynamic-get
dynamic-put
edit
eq
eval
evenp
exit
exp
expand$
explode$
fact-existp
fact-index
fact-relation
fact-slot-names
fact-slot-value
facts
fetch
find-all-instances
find-instance
first$
float
floatp
focus
format
gensym
gensym*
get
get-auto-float-dividend
get-current-module
get-defclass-list
get-deffacts-list

get-deffunction-list
get-defgeneric-list
get-defglobal-list
get-definstances-list
get-defmessage-handler-list
get-defmethod-list
get-defmodule-list
get-defrule-list
get-deftemplate-list
get-dynamic-constraint-checking
get-fact-duplication
get-fact-list
get-focus
get-focus-stack
get-function-restrictions
get-incremental-reset
get-method-restrictions
get-reset-globals
get-salience-evaluation
get-sequence-operator-recognition
get-static-constraint-checking
get-strategy
grad-deg
halt
if
implode$
init-slots
initialize-instance
initialize-instance
insert$
instance-address
instance-addressp
instance-existp
instance-name
instance-name-to-symbol
instance-namep
instancep
instances
integer
integerp
length
length$
lexemep

list-defclasses
list-deffacts
list-deffunctions
list-defgenerics
list-defglobals
list-definstances
list-defmessage-handlers
list-defmethods
list-defmodules
list-defrules
list-deftemplates
list-focus-stack
list-watch-items
load
load*
load-facts
load-instances
log
log10
loop-for-count
lowcase
make-instance
make-instance
matches
max
mem-requests
mem-used
member
member$
message-duplicate-instance
message-duplicate-instance
message-handler-existp
message-modify-instance
message-modify-instance
min
mod
modify
modify-instance
modify-instance
multifieldp
mv-append
mv-delete

mv-replace
mv-slot-delete
mv-slot-insert
mv-slot-replace
mv-subseq
neq
next-handlerp
next-methodp
not
nth
nth$
numberp
object-pattern-match-delay
oddp
open
options
or
override-next-handler
override-next-method
pi
pointerp
pop-focus
ppdefclass
ppdeffacts
ppdeffunction
ppdefgeneric
ppdefglobal
ppdefinstances
ppdefmessage-handler
ppdefmethod
ppdefmodule
ppdefrule
ppdeftemplate
ppinstance
preview-generic
preview-send
primitives-info
print-region
printout
progn
progn$
put
rad-deg

random
read
readline
refresh
refresh-agenda
release-mem
remove
remove-break
rename
replace$
reset
rest$
restore-instances
retract
return
round
rule-complexity
rules
run
save
save-facts
save-instances
sec
sech
seed
send
sequencep
set-auto-float-dividend
set-break
set-current-module
set-dynamic-constraint-checking
set-fact-duplication
set-incremental-reset
set-reset-globals
set-salience-evaluation
set-sequence-operator-recognition
set-static-constraint-checking
set-strategy
setgen
show-breaks
show-defglobals
show-fht

show-fpn
show-joins
show-opn
sin
sinh
slot-allowed-values
slot-cardinality
slot-delete$
slot-direct-accessp
slot-direct-delete$
slot-direct-insert$
slot-direct-replace$
slot-existp
slot-facets
slot-initablep
slot-insert$
slot-publicp
slot-range
slot-replace$
slot-sources
slot-types
slot-writablep
sqrt
str-assert
str-cat
str-compare
str-explode
str-implode
str-index
str-length
stringp
sub-string
subclassp
subseq$
subset
subsetp
superclassp
switch
sym-cat
symbol-to-instance-name
symbolp
system
tan

tanh
time
toss
type
type
undefclass
undeffacts
undeffunction
undefgeneric
undefglobal
undefinstances
undefmessage-handler
undefmethod
undefrule
undeftemplate
unmake-instance
unwatch
upcase
watch
while
wordp

Index