Jess® The Rule Engine for the Java™ Platform

Version 7.1p2

November 5, 2008

Table of Contents

Jess® *the Rule Engine for the Java™ Platform*

Introduction

Version 7.1p2 (5 November 2008) DRAFT
Ernest J. Friedman-Hill
Sandia National Laboratories

Jess is a *rule engine* for the Java platform. To use it, you specify logic in the form of rules using one of two formats: the Jess rule language (prefered) or XML. You also provide some of your own data for the rules to operate on. When you run the rule engine, your rules are carried out. Rules can create new data, or they can do anything that the Java programming language can do.

Although Jess can run as a standalone program, usually you will embed the Jess library in your Java code and manipulate it using its own Java API or the basic facilities offered by the javax.rules API.

You can develop Jess language code in any text editor, but Jess comes with a full featured development environment based on the award-winning Eclipse platform.

Hyperlinks:

The hyperlinks in the Table of Contents link to the chapters and sections in this MS Word document. All other hyperlinks within the document link to the HTML document pages. You should place this document in the same folder (docs) as the HTML files or modify the Hyperlink base field in File **à** Properties **à** Summary. This document was created using MS Word 2002 using MS Windows XP Pro. The document was stored in the directory D:\Jess71p2\docs. (At times, bitmaps disappear if the document is not placed in the directory D:\Jess71p2\docs)

1. Getting Started

1.1. Requirements

Jess is a programmer's library written in Java. Therefore, to use Jess, you'll need a Java Virtual Machine (JVM). You can get an excellent JVM for Windows, Linux, and Solaris free from Sun Microsystems. Jess 7.1 is compatible with all released versions of Java starting with JDK 1.4, including JDK 1.6, the latest release. Older Jess versions numbered 4.x were compatible with JDK 1.0, 5.x versions worked with JDK 1.1, and Jess 6 worked with JDK 1.2 and up.

Be sure your JVM is installed and working correctly before trying to use Jess.

To use the JessDE integrated development environment, you'll need version 3.1 or later of the Eclipse SDK from http://www.eclipse.org. Be sure that Eclipse is installed and working properly before installing the JessDE.

The Jess library serves as an interpreter for another language, which I will refer to in this document as the Jess language. The Jess language is a highly specialized form of Lisp.

I am going to assume that you, the reader, are a programmer who will be using either one or both of these languages. I will assume that all readers have at least a minimal facility with Java. You must have a Java runtime system, and you must know how to use it at least in a simple way. You should know how to use it to

- run a Java application
- deal with configuration issues like the CLASSPATH variable
- (optional) compile a collection of Java source files

If you do not have at least this passing familiarity with a Java environment, then may I suggest you purchase an introductory book on the topic. Java software for many platforms -- as well as a wealth of tutorials and documentation -- is available at no cost from http://java.sun.com.

For those readers who are going to program in the Jess language, I assume general familiarity with the principles of programming. I will describe the entire Jess language, so no familiarity with Lisp is required (although some is helpful.) Furthermore, I will attempt to describe, to the extent possible, the important concepts of rule-based systems as they apply to Jess. Again, though, I will assume that the reader has some familiarity with these concepts and more. If you are unfamiliar with rule-based systems, you may want to purchase a text on this topic as well.

Many readers will want to extend Jess' capabilities by either adding commands (written in Java) to the Jess language, or embedding the Jess library in a Java application. Others will want to use the Jess language's Java integration capabilities to call Java functions from Jess language programs. In sections of this document targeted towards these readers, I will assume moderate knowledge of Java programming. I will not teach any aspects of the Java language. The interested reader is once again referred to your local bookstore.

This document contains a bibliography wherein a number of books on all these topics are listed.

1.2. Getting ready

1.2.1. Unpacking the Distribution

Jess is supplied as a single .zip file which can be used on all supported platforms. This single file contains all you need to use Jess on Windows, UNIX, or the Macintosh (except for a JVM, which you must install yourself.)
When Jess is unpacked, you should have a directory named `Jess71p2/`. Inside this directory should be the following files and subdirectories:

`README`	A quick start guide.
`LICENSE`	Information about your rights regarding the use of Jess.
`bin`	A directory containing a Windows batch file (jess.bat) and a UNIX shell script (jess) that you can use to start the Jess command prompt.
`lib`	A directory containing Jess itself, as a Java archive file. Note that this is *not* a "clickable" archive file; you can't double-click on it to run Jess. This is deliberate. This directory also contains the JSR-94 (javax.rules) API in the file jsr94.jar.
`docs/`	This documentation. "index.html" is the entry point for the Jess manual.
`examples/jess`	A directory of small example programs in the Jess language.
`examples/xml`	A directory of small example programs in JessML, Jess's XML rule language.
`eclipse`	The JessDE, Jess's Integrated Development Environment, supplied as a set of plugins for Eclipse 3.0. See here for installation instructions.
`src` (Optional)	If this directory is present, it contains the full source for the Jess rule engine and development environment, including an Ant script for building it.

1.2.2. Command-line Interface

Jess has an interactive command-line interface. The distribution includes two scripts that you can run to get a Jess command prompt: one for Windows, and one for UNIX. They're both in the bin/ directory. Run the one that's appropriate for your system, and you should see something like this

```
C:\Jess71p2> bin\jess.bat

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p1 8/6/2008

Jess>
```

That's the Jess prompt. Try evaluating a prefix math expression like "(+ 2 2)". Don't forget those parentheses!

```
Jess> (+ 2 2)

4
```

Jess evaluates this function, and prints the result. In the next chapter of this manual, we'll look at the syntax of the Jess rule language itself.
To execute a file of Jess code from the Jess prompt, use the `batch` command:

```
Jess> (batch "examples/jess/sticks.clp")
```

```
Who moves first (Computer: c Human: h)?
```
Note that in the preceding example, you type what follows the `Jess>` prompt, and Jess responds with the text on the next line. I will follow this convention throughout this manual.
To execute the same Jess program directly from the operating-system prompt, you can pass the name of the program as an argument to the script that starts Jess:

```
C:\Jess71p2> bin\jess.bat examples\jess\sticks.clp

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p1 8/6/2008

Who moves first (Computer: c Human: h)?
```

1.2.2.1. Command-line editing

When working at the Jess command line, it's convenient to be able to edit and reinvoke previous commands. The Jess library doesn't support this directly, but the excellent open-source product JLine can add this capability to any command-line Java program, Jess included. JLine works great with the Jess prompt and I strongly recommend it.

1.2.2.2. Graphical console

The class `jess.Console` is a simple graphical version of the Jess command-line interface. You type into a text field at the bottom of the window, and output appears in a scrolling window above. Type

```
C:\Jess71p2> java -classpath lib\jess.jar jess.Console
```
from the Jess71p2 directory to try it.

1.2.3. Java Programming with Jess

To use Jess as a library from your Java programs, the file `jess.jar` (in the lib directory) must either be on your class path, be installed as a standard extension, or your development tools must be set up to recognize it. The details of doing these tasks are system and environment dependent, but setting the class path usually involves modifying an environment variable, and installing a standard extension simply means copying `jess.jar` into your `$(JAVA_HOME)/jre/lib/ext` directory. Refer to the Java documentation or an introductory Java text for details.

1.2.4. Jess Example Programs

There are some simple example programs (in the examples/jess and examples/xml directories) that you can use to test that you have installed Jess properly. These include `fullmab.clp`, `zebra.clp`, and `wordgame.clp`. `fullmab.clp` is a version of the classic Monkey and Bananas problem. To run it yourself from the command line, just type:

```
C:\Jess71p2> bin\jess examples\jess\fullmab.clp
```
and the problem should run, producing a few screens of output. Any file of Jess code can be run this way. Giving Jess a file name on the command line is like using the `batch` function. Therefore, you generally need to make sure that the file ends with:

```
Jess> (reset)
(run)
```

or no rules will fire. The `zebra.clp` and `wordgame.clp` programs are two classic examples selected to show how Jess deals with tough situations. These examples both generate large numbers of partial pattern matches, so they are slow and use up a lot of memory. Other examples include `sticks.clp` (an interactive game) and `jframe.clp` (a demo of building a graphical interface using Jess's Java integration capabilities).

The XML examples are in individual subdirectories; each subdirectory contains a README file with instructions for running that example.

2. The JessDE Developer's Environment

Jess 7 includes an Eclipse-based development environment. There is an editor, a debugger, and a Rete network viewer. More components (a rule browser and other tools) will be included in future releases.

2.1. Installing the JessDE

The Jess Developer's Environment (JessDE) is supplied as a set of plugins for the popular open-source IDE Eclipse; in particular, these are plugins for Eclipse version 3.1 or later. Note that the JessDE works only with the full "Eclipse SDK" -- the smaller "Platform Runtime Binary" is insufficient.

To install the JessDE, simply exit Eclipse, unzip all the files from Jess71p2/eclipse into the top-level Eclipse installation directory. Confirm that a directory named "plugins/gov.sandia.jess_7.1.0" exists under your Eclipse installation directory, and then restart Eclipse.

IMPORTANT! If you're updating from a previous version of the JessDE, you must launch Eclipse with the "-clean" command-line switch to force it to update the information it caches regarding the JessDE plugins. If you don't do this, many of the JessDE's options may be disabled. You only need to do this once after the installation.

2.1.1. Verifying your installation

Under the "Help" menu, choose "about Eclipse SDK". There will be a button with the Jess logo on it on the main "About Eclipse SDK" window. Press "Plug-in Details". If the JessDE is installed properly, you'll find three or four Jess-related plugins in the list -- in my copy of Eclipse, they appear near the bottom.

Then create a Java project using the "New Project" wizard. Create a new file in that project, and name it "hello.clp". It should open in the Jess editor, which has a little silver ball with a red "J" as its icon. Enter some Jess code, like

```
(printout t "Hello, World" crlf)
```

You should see appropriate syntax highlighting. If so, congratulations, everything is working! Read on for more information about the other features of the JessDE.

2.1.2. A few more details

The JessDE editor creates problem markers to point out syntax errors and warnings in your Jess documents. You most likely want to have these markers show up in the Eclipse "Problems" view, although they might not show up by default. After installing the JessDE and restarting Eclipse, in the Problems view, click on the "Filters" icon in the title bar, and check the box labelled "Jess Problem" (if it's not already checked.) Your Problems view should now display Jess errors and warnings.

To use the Rete Network View, you'll need to have the Eclipse Graph Editing Framework (GEF) installed. You can get the GEF from the Eclipse project page here, or install it through Eclipse's built-in update manager. Then to display this view, find it under the "Jess Debugger" group in Eclipse's "Show view" dialog. Then when the cursor is inside a rule in a Jess editor window, the Rete Network View will show the compiled network for that rule.

2.2. Using the JessDE

2.2.1. The Jess language editor

The JessDE editor can edit ".clp" files. By default, any file you create with the extension "clp" will be opened using the JessDE editor. There's no separate Jess perspective, or Jess project type; we expect that most people will use the JessDE tools to write the Jess components of a hybrid Jess/Java application, and so the tools expect to be used on files in a Java project. The JessDE uses your Java project's class path to resolve Java class names used in Jess language code -- i.e., in a call to the `defclass` function.

The editor has all the features you'd expect from a modern programmer's editor.

Syntax coloring you can customize.
: You can change the default colors using the "Jess Editor" tab of the Eclipse global preferences dialog.

Content assistant supplies deftemplate, slot and function names.
: You can invoke Eclipse's "Content Assist" feature in many different places within the JessDE editor; JessDE will make it much easier to enter Jess code. Press Alt-'/' while typing to produce a list of choices.

"Quick fix" assistant can repair code automatically
: This feature is bound to Ctrl-1 by default. Quick fix currently knows how to define undefined deftemplates, and add new slots to existing deftemplates (if they're defined in the same file.) More to come!

Real-time error checking with markers and error highlighting
: Errors and warnings are highlighted as you type

Automatic code formatting
: Code is indented as you type. You can also choose "Format" from the "Source" menu to format an entire buffer.

Fast navigation using outline view
: The Eclipse outline view lists all the constructs defined in a buffer; you can click on any one of them to quickly navigate to it.

Parenthesis matching and auto-insertion
: When you type a '(' or '"' character, JessDE insert the matching character as well. When your cursor is next to a parenthesis, JessDE shows you the matching parenthesis.

Online help for Jess functions and constructs via "hovers"
: You have quick access to the Jess manual description of every function and construct type.

Help hovers for deftemplates and deffunctions
: If you hold your mouse over the name of a deftemplate or deffunction, anywhere in the code, JessDE will show a "tooltip" containing information about that template or function.

Run and Debug commands for Jess programs
: You can run or debug a Jess program using the normal Eclipse "Run..." menu or by right clicking on Navigator items or in the edit window.

2.2.2. Dependencies among files

Sometimes one `*.clp` file depends on code in some other `*.clp` file having been read first; for example, `rules.clp` might need the definitions in `templates.clp`. Without these definitions, `rules.clp` will appear to have syntax errors. To deal with this, you can use the `require*` function. "require*" lets you explicitly declare these dependencies.

If a file `rules.clp` depends on Jess commands being executed from Java, you can deal with this by creating a special file just for this purpose (you might call it `ruledepends.clp`). That special file can contain whatever declarations are needed to make `rule.clp` parse properly in the editor. If you add "`(require* ruledepends)`" to `rules.clp`, then this extra file will be parsed only if it's present, as it will be during development. When you deploy the code, you can simply not deploy `ruledepends.clp`, and allow `rules.clp` to get its declarations from Java code.

The "require" mechanism replaces the "Source dependencies" property sheet from earlier versions of JessDE, which is no longer supported.

2.2.3. The Rete Network view

You can instantly see a graphical representation of the Rete network derived from any rule using the JessDE's "Rete Network View". When this view is open (you can open it using the "Windows | View | Other..." dialog in Eclipse) it will display the Rete network for the rule that the editor caret is in. You can use this to see, in real time, how modifying a rule changes the Rete network. The graph layout is far superior to that you get from the Jess "view" command -- there are no overlapping or crossing lines, and the height of each "column" can vary.

2.2.4. The Jess debugger

The JessDE debugger lets you debug a Jess program defined in a .clp file. It has all the features you'd expect from a graphical debugger: you can suspend and resume a program, or step through it. When the program is stopped, the contents of the execution stack are displayed, and you can examine the variables defined in each stack frame. Selecting a stack frame also navigates to the source code being executed. You can also set (or clear) breakpoints in any .clp file by right-clicking in the ruler on the left-hand edge of the editor window. Breakpoints can be set only on functions (either built-in or user defined), so you can't break on a `defrule` or `deftemplate` construct. You can, however, break on a function call on the left or right-hand side of a rule.

3. Jess Language Basics

Most of the time, you'll write Jess rules in the *Jess rule language*. If you've never used Lisp, the Jess rule language may look a bit odd at first, but it doesn't take long to learn. The payoff is that it's very expressive, and can implement complex logical relationships with very little code.

In this chapter, we'll look at the basic syntax of the Jess language. In subsequent chapters, we'll learn how to define high-level concepts like *facts* and *rules,* but here, we'll just be looking at the nuts and bolts.

In this language guide, I'll use an extremely informal notation to describe syntax. Basically strings in <angle-brackets> are some kind of data that must be supplied; things in [square brackets] are optional, things ending with `+` can appear one or more times, and things ending with `*` can appear zero or more times. In general, input to Jess is free-format. Newlines are generally not significant and are treated as whitespace; exceptions will be noted.

3.1. Symbols

The symbol is a core concept of the Jess language. Symbols are very much like identifiers in other languages. A Jess symbol can contain letters, digits, and the following punctuation: `$*=+/<>_?#.` . A symbol may not begin with a digit; it may begin with some punctuation marks (some have special meanings as operators when they appear at the start of a symbol).
Jess symbols are case sensitive: `foo`, `FOO` and `Foo` are all different symbols.
The best symbols consist of letters, digits, underscores, and dashes; dashes are traditional word separators. The following are all valid symbols:

```
foo first-value contestant#1 _abc
```

There are three "magic" symbols that Jess interprets specially: `nil`, which is somewhat akin to Java's `null` value; and `TRUE` and `FALSE`, which are Jess' boolean values.

3.2. Numbers

Jess uses the Java functions `parseInt(java.lang.String)`, `parseLong(java.lang.String)`, and `parseDouble(java.lang.String)` to parse integer, long, and floating point numbers, respectively. See the documentation for those methods for a precise syntax description. The following are all valid numbers:

```
3 4. 5.643 5654L 6.0E4 1D
```

3.3. Strings

Character strings in Jess are denoted using double quotes (`"`). Backslashes (`\`) can be used to escape embedded quote symbols. Note that Jess strings are unlike Java strings in several important ways. First, no "escape sequences" are recognized. You cannot embed a newline in a string using "`\n`", for example. On the other hand, real newlines are allowed inside double-quoted strings; they become part of the string. The following are all valid strings:

```
"foo" "Hello, World" "\"Nonsense,\" he said firmly." "Hello,
```

```
There"
```
The last string is equivalent to the Java string "Hello,\nThere".

3.4. Lists

Another fundamental unit of syntax in Jess is the list. A list always consists of an enclosing set of parentheses and zero or more symbols, numbers, strings, or other lists. The following are valid lists:

```
(+ 3 2) (a b c) ("Hello, World") () (deftemplate foo (slot bar))
```
The first element of a list (the *car* of the list in Lisp parlance) is often called the list's *head* in Jess.

3.5. Comments

Jess supports two kinds of programmer's comments: Lisp-style line comments and C-style block comments. Line comments begin with a semicolon (`;`) and extend to the end of the line of text. Here is an example of a line comment:

```
; This is a list
(a b c)
```
Block comments work as they do in C: they start with the two characters "/*" and end with "*/". Block comments don't nest.

```
/*
   Here is an example of a list (commented out):
   (a b c)
*/
```
Comments can appear anywhere in a Jess program, including inside constructs like templates and rules.

3.6. Calling functions

As in Lisp, all code in Jess (control structures, assignments, procedure calls) takes the form of a function call. There are no "operators"; *everything* is a function call. However, some functions have names that look like Java operators, and in these cases, they operate much like their Java counterparts.

Function calls in Jess are simply lists. Function calls use a prefix notation; a list whose head is a symbol that is the name of an existing function can be a function call. For example, an expression that uses the `+` function to add the numbers `2` and `3` would be written `(+ 2 3)`. When evaluated, the value of this expression is the number `5` (not a list containing the single element `5`!). In general, expressions are recognized as such and evaluated in context when appropriate. You can type expressions at the `Jess>` prompt. Jess evaluates the expression and prints the result:

```
Jess> (+ 2 3)

5

Jess> (+ (+ 2 3) (* 3 3))

14
```

Note that you can nest function calls; the outer function is responsible for evaluating the inner function calls.
Jess comes with a large number of built-in functions that do everything from math, program control and string manipulations, to giving you access to Java APIs. You can also define your own functions either in the Jess language or in Java.
One of the most commonly used functions is `printout`, which is used to send text to Jess's standard output, or to a file. A complete explanation will have to wait, but for now, all you need to know is contained in the following example:

```
Jess> (printout t "The answer is " 42 "!" crlf)

The answer is 42!
```

Another useful function is `batch`, which evaluates a file of Jess code. To run the Jess source file `examples/jess/hello.clp` you can enter

```
Jess> (batch "examples/jess/hello.clp")

Hello, world!
```

Each of these functions (along with all the others) is described more thoroughly in the Jess function guide.

3.7. Variables

Programming variables in Jess are identifiers that begin with the question mark (`?`) character. The question mark is part of the variable's name. The name can contain any combination of letters, numbers, dash (-), underscore(_), colon (:) or asterisk (*) characters. Variable names may *not* contain a period (.) .
A variable can refer to a single symbol, number, or string, or it can refer to a list.
You can assign a value to to a variable using the `bind` function:

```
Jess> (bind ?x "The value")

"The value"
```

Variables need not (and cannot) be declared before their first use (except for special variables called defglobals).
To see the value of a variable at the `Jess>` prompt, you can simply type the variable's name.

```
Jess> (bind ?a 123)

123

Jess> ?a

123
```

3.7.1. Dotted variables

New in Jess 7.1 are *dotted variables.* A dotted variable looks like `?x.y`. Jess always interprets this as referring to the slot `y` of the fact in variable `?x`. You can use dotted variables in any procedural code, but they won't generally work in the pattern matching parts of a rule.

Reading the value of a dotted variable results in a call to `fact-slot-value`, while using `bind` to set such a variable will result in a call to `modify`. Dotted variables are a great convenience and can make a lot of Jess code read more clearly.

3.7.2. Global variables (or defglobals)

Any variables you create at the `Jess>` prompt, or at the "top level" of any Jess language program, are cleared whenever the `reset` command is issued. This makes them somewhat transient; they are fine for scratch variables but are not persistent global variables in the normal sense of the word. To create global variables that are not destroyed by `reset`, you can use the `defglobal` construct.

```
(defglobal [?<global-name> = <value>]+)
```

Global variable names must begin and end with an asterisk. Valid global variable names look like

```
?*a*    ?*all-values*    ?*counter*
```

When a global variable is created, it is initialized to the given value. When the `reset` command is subsequently issued, the variable *may* be reset to this same value, depending on the current setting of the `reset-globals` property. There is a function named `set-reset-globals` that you can use to set this property. An example will help.

```
Jess> (defglobal ?*x* = 3)
TRUE
Jess> ?*x*
3
Jess> (bind ?*x* 4)
4
Jess> ?*x*
4
Jess> (reset)
TRUE
Jess> ?*x*
3
Jess> (bind ?*x* 4)
4
Jess> (set-reset-globals nil)
FALSE
Jess> (reset)
TRUE
```

```
Jess> ?*x*

4
```

You can read about the `set-reset-globals` and the accompanying `get-reset-globals` function in the Jess function guide.

3.8. Control flow

In Java, control flow -- branching and looping, exception handling, etc -- is handled by special syntax and keywords like `if`, `while`, `for`, and `try`. In Jess, as we said before, everything is a function call, and control flow is no exception. Therefore, Jess includes functions named `if`, `while`, `for`, and `try`, along with others like `foreach`. Each of these functions works similarly to the Java construct of the same name.

3.8.1. A simple loop

For example, a Jess "while" loop looks like this:

```
Jess> (bind ?i 3)

3

Jess> (while (> ?i 0)
          (printout t ?i crlf)
          (-- ?i))

3
2
1
FALSE
```

The first argument to `while` is a boolean expression. The while function evaluates its first argument and, if it is true, evaluates all its other arguments. It repeats this procedure until the first argument evaluates to `FALSE`; a while loop always returns FALSE.

There are several other looping functions built into Jess; see the descriptions of `for` and `foreach` in the Jess function index. There is also a `break` function that can be used to abort loops as well as return early from the right-hand-side of a rule.

3.8.2. Decisions and branching

The `if` function looks like this:

```
Jess> (bind ?x 1)

1

Jess> (if (> ?x 100) then
          (printout t "X is big" crlf)
       elif (> ?x 50) then
              (printout t "X is medium" crlf)
       else
          (printout t "X is small" crlf))
```

```
X is small
```

Again, the first argument is a Boolean expression, and the second is always the symbol `then`. If the expression is not `FALSE`, `if` will execute the remaining arguments up until it sees of of the the (optional) symbols `elif` or `else`. If there are one or more`elif`s, then their Boolean expressions control whether their actions will be executed instead. If `else` appears, then any arguments following it are evaluated if all the Boolean expressions are `FALSE`.

4. Defining Functions in Jess

4.1. Deffunctions

You can define your own functions in the Jess rule language using the `deffunction` construct. A `deffunction` construct looks like this:

```
(deffunction <function-name> [<doc-comment>] (<parameter>*)
   <expr>* [<return-specifier>])
```

The `<function-name>` must be a symbol. Each `<parameter>` must be a variable name. The optional `<doc-comment>` is a double-quoted string that can describe the purpose of the function. There may be an arbitrary number of `<expr>` expressions. The optional `<return-specifier>` gives the return value of the function. It can either be an explicit use of the `return` function or it can be any value or expression. Control flow in `deffunction`s is achieved via control-flow functions like `foreach`, `if`, and `while`. The following is a `deffunction` that returns the larger of its two numeric arguments:

```
Jess> (deffunction max (?a ?b)
  (if (> ?a ?b) then
      (return ?a)
  else
      (return ?b)))

TRUE
```

Note that this could have also been written as:

```
Jess> (deffunction max (?a ?b)
  (if (> ?a ?b) then
      ?a
   else
      ?b))

TRUE
```

This function can now be called anywhere a Jess function call can be used. For example

```
Jess> (printout t "The greater of 3 and 5 is " (max 3 5) "." crlf)

The greater of 3 and 5 is 5.
```

Normally a `deffunction` takes a specific number of arguments. To write a `deffunction` that takes an arbitrary number of arguments, make the last formal parameter be a *multifield* -- a variable prefixed with a '$' character. When the `deffunction` is called, the multifield variable will contain all the remaining arguments passed to the function, as a list. A `deffunction` can accept no more than one such wildcard argument, and it must be the last argument to the function.

You can also customize the Jess language with functions written in Java. These are indistinguishable from built-in functions, and in fact, you write them using the same interface used to define built-in functions. See here for details.

4.2. Defadvice

Sometimes a Jess function won't behave exactly as you'd like. The `defadvice` construct lets you write some Jess code which will be executed before or after each time a given Jess function is called. `defadvice` lets you easily "wrap" extra code around any Jess function, such that it

executes before (and thus can alter the argument list seen by the real function, or short-circuit it completely by returning a value of its own) or after the real function (and thus can see the return value of the real function and possibly alter it.) defadvice provides a great way for Jess add-on authors to extend Jess without needing to change any internal code.
Here are some examples of what defadvice looks like.
This intercepts calls to 'plus' (+) and adds the extra argument '1', such that (+ 2 2) becomes (+ 2 2 1) -> 5. The variable '$?argv' is special. It always refers to the list of arguments the real Jess function will receive when it is called.

```
Jess> (defadvice before + (bind $?argv (create$ $?argv 1)))

TRUE

Jess> (+ 2 2)

5
```

This makes all additions equal to 1. By returning, the defadvice keeps the real function from ever being called.

```
Jess> (defadvice before + (return 1))

TRUE

Jess> (+ 2 2)

1
```

This subtracts one from the return value of the + function. ?retval is another magic variable - it's the value the real function returned. When we're done, we remove the advice with `undefadvice`.

```
Jess> (defadvice after + (return (- ?retval 1)))

TRUE

Jess> (+ 2 2)

3

Jess> (undefadvice +)

Jess> (+ 2 2)

4
```

4. Defining Functions in Jess

5. Working Memory

Each Jess rule engine holds a collection of knowledge nuggets called *facts.* This collection is known as the *working memory.* Working memory is important because rules can only react to additions, deletions, and changes to working memory. You can't write a Jess rule that will react to anything else.

Some facts are *pure facts* defined and created entirely by Jess. Other facts are *shadow facts* connected to Java objects you provide. Shadow facts act as "bridges" that let Jess reason about things that happen outside of working memory.

Every fact has a *template*. The template has a name and a set of *slots*, and each fact gets these things from its template. This is the same structure that JavaBeans -- plain old Java objects, or POJOs -- have, and it's also similar to how relational databases are set up. The template is like the class of a Java object, or like a relational database table. The slots are like the properties of the JavaBean, or the columns of a table. A fact is therefore like a single JavaBean, or like a row in a database table. You can think of it either way.

In Jess, there are three kinds of facts: *unordered facts*, *shadow facts* and *ordered facts*. We'll learn about all of these in this chapter. First, though, we need to learn more about templates.

Just so you know, as we learn about working memory: you can see a list of all the facts in working memory using the `facts` command. Facts are added using the `assert`, `add`, and `definstance` functions. You can remove facts with the `retract` and `undefinstance` functions. The `modify` function lets you change the slot values of facts already in working memory. And finally, you can completely clear Jess of all facts and other data using the `clear` command.

5.1. Templates

As we've already stated, every fact has a *template*. A fact gets its *name* and its list of *slots* from its template. Therefore a template is something like a Java class.

You usually create templates yourself using the `deftemplate` construct or the `defclass` function. Other times, templates are created automatically for you, either while you're defining a `defrule` or when you use the `add` function or the `jess.Rete.add(java.lang.Object)` method.

The deftemplate construct is the most general and most powerful way to create a template. You won't understand most of the options shown here yet, but we'll cover them all in this chapter and the next:

```
(deftemplate template-name
    [extends template-name]
    ["Documentation comment"]
    [(declare  (slot-specific TRUE | FALSE)
               (backchain-reactive TRUE | FALSE)
               (from-class class name)
               (include-variables TRUE | FALSE)
               (ordered TRUE | FALSE))]
     (slot | multislot slot-name
```

```
        ([(type ANY | INTEGER | FLOAT |
               NUMBER | SYMBOL | STRING |
               LEXEME | OBJECT | LONG)]
        [(default default value)]
        [(default-dynamic expression)]
        [(allowed-values expression+)])*)
```

A template declaration includes a name, an optional documentation string, an optional "extends" clause, an optional list of declarations, and a list of zero or more slot descriptions. Each slot description can optionally include a type qualifier or a default value qualifier. In the syntax diagram, defaults for various options are indicated in bold letters.

The *template-name* is the head of the facts that will be created using this template.

The declarations affect either how the template will be created, or how facts that use it will act, or both. We'll cover most of the declarations in this chapter; others are covered in the Constructs appendix or in the chapter on rules.

Every template has a single parent template, which can be specified with the "extends" clause. A template inherits all the slots of its parent template, as well as declared properties like `slot-specific` and `backchain-reactive`. If you don't specify a parent, a template extends a template named "__fact", which has no slots.

Some template declarations include *slots* (those that don't will generally have implicitly defined slots.) There may be an arbitrary number of slots in a template. Each <*slot-name*> must be a symbol. A *multislot* is a slot that can hold a list of values, while a normal slot can hold just one value at a time. The name of a slot may *not* contain a '.' (dot) character.

Each slot can have a list of zero or more *slot qualifiers.* The `default` slot qualifier gives a value to use for a slot when the fact is first created, if no other value is specified; the default is the symbol `nil` for a regular slot, and the empty list for a multislot. `default-dynamic` is similar but the the given *expression* will be evaluated each time a new fact using this template is asserted.

The `type` slot qualifier is accepted but not currently enforced by Jess; in theory it specifies what data type the slot is allowed to hold. Acceptable values are ANY, INTEGER, FLOAT, NUMBER, SYMBOL, STRING, LEXEME, and OBJECT.

The `allowed-values` slot qualifier gives a set of values allowed to be in the slot. For a multislot, the allowed values are restricted to values in this set. If you specify both `allowed-values` and `default`, Jess checks to make sure they're consistent. If you specify `allowed-values` but do not specify a slot default, then the first listed allowed value becomes the default.

5.1.1. Undefining templates

Although it shouldn't be a common requirement, you can remove a previously defined template using the `jess.Rete.removeDeftemplate(String)` method. You might want to do this during interactive development, because you can't change the slots of a template without removing the old definition first. You won't be able to remove a template unless it's completely unused: no rules, other templates, deffacts, or facts may reference it.

5.2. Unordered facts

In object-oriented languages like Java, *objects* have named *fields* in which data appears. Unordered facts offer this capability (although the fields are traditionally called *slots.*)

```
(automobile (make Ford) (model Explorer) (year 1999))
```

Before you can create unordered facts, you have to define the slots they have using the `deftemplate` construct.

As an example, defining the following template:

```
Jess> (deftemplate automobile
  "A specific car."
  (slot make)
  (slot model)
  (slot year (type INTEGER))
  (slot color (default white)))
```

would allow you to define the fact shown here.

```
Jess> (reset)

Jess> (assert (automobile (model LeBaron) (make Chrysler)
        (year 1997)))

<Fact-1>

Jess> (facts)


f-0   (MAIN::initial-fact)
f-1   (MAIN::automobile (make Chrysler) (model LeBaron)
                        (year 1997) (color white))
For a total of 2 facts in module MAIN.
```

Note that the car is white by default. If you don't supply a default value for a slot, and then don't supply a value when a fact is asserted, the special value `nil` is used. Also note that any number of additional automobiles could also be simultaneously asserted onto the fact list using this template.

Note also that we can specify the slots of an unordered fact in any order (hence the name.) Jess rearranges our inputs into a *canonical order* so that they're always the same.

As you can see above, each fact is assigned an integer index (the *fact-id*) when it is asserted. You can remove an individual fact from the working memory using the `retract` function.

```
Jess> (retract 1)

TRUE
```

```
Jess> (facts)

f-0   (MAIN::initial-fact)
For a total of 1 facts in module MAIN.
```

The fact `(initial-fact)` is asserted by the `reset` command. It is used internally by Jess to keep track of its own operations; you should generally not retract it.

A given slot in a deftemplate fact can normally hold only one value. If you want a slot that can hold multiple values, use the `multislot` keyword instead:

```
Jess> (deftemplate box (slot location) (multislot contents))

TRUE

Jess> (bind ?id (assert (box (location kitchen)
                       (contents spatula sponge frying-pan))))

<Fact-2>
```

(We're saving the fact returned by (assert) in the variable ?id, for use below.) A multislot has the default value () (the empty list) if no other default is specified.

You can change the values in the slots of an unordered fact using the `modify` command. Building on the immediately preceding example, we can move the box into the dining room:

```
Jess> (modify ?id (location dining-room))

<Fact-2>

Jess> (facts)

f-0   (MAIN::initial-fact)
f-2   (MAIN::box (location dining-room)
                 (contents spatula sponge frying-pan))
For a total of 2 facts in module MAIN.
```

The optional `extends` clause of the `deftemplate` construct lets you define one template in terms of another. For example, you could define a used-auto as a kind of automobile with more data:

```
Jess> (deftemplate used-auto extends automobile
  (slot mileage)
  (slot blue-book-value)
  (multislot owners))

TRUE
```

A used-auto fact would now have all the slots of an automobile, in the same order, plus three more. As we'll see later, this inheritance relationship will let you act on all automobiles (used or not) when you so desire, or only on the used ones.

5.3. Shadow facts: reasoning about Java objects

As mentioned previously, shadow facts are just unordered facts that serve as "bridges" to Java objects. By using shadow facts, you can put any Java object into Jess's working memory.

5.3.1. Templates for shadow facts

Like all other facts, shadow facts need to have a template, In this case, though, rather than specifying the slots ourselves, we want to let Jess create the template automatically by looking at a Java class. For example, we might be writing a banking program. Our imaginary Java code works with `Account` objects, like this:

```
import java.io.Serializable;

public class Account implements Serializable {
    private float balance;
    public float getBalance() { return balance; }
    public void setBalance(float balance) {
        this.balance = balance;
    }
    // Other, more interesting methods
}
```

At some point the rule-based part of our program needs to deal with these too. Therefore we'll need a template like this:

```
Jess> (deftemplate Account
    (declare (from-class Account)))
```

Note how I've used the class name (minus the package prefix, if there was one) as the template name. This is just a convention, and the template name can be anything you want. But as we'll see a little later, Jess knows about this convention and using it can save a little work.

This template will automatically have slots corresponding to the the JavaBeans properties of the `Account` class: in particular, there will be a slot named `balance` corresponding to the `getBalance()` method. Jess uses the `java.beans.Introspector` class to find the property names, so you can customize how Jess defines properties by writing your own `java.beans.BeanInfo` classes.

The `defclass` function can be used instead to create a shadow fact template; it looks like

```
Jess> (defclass Account Account)
```

It is just a shortcut for using deftemplate. The chief advantage of `defclass` is that because it is a function, it can be used anywhere, unlike `deftemplate` which is a construct, and can only be used at the top level of a program (see here for more information.) The chief disadvantage is that it is limited in what it can do: many of the declarables that are available with `deftemplate` are not available with `defclass`.

The `from-class` declaration, by itself, will create slots that correspond to JavaBeans properties. If you also use `include-variables`, like this:

```
Jess> (deftemplate Account
    (declare (from-class Account)
             (include-variables TRUE))
```

then public member variables of the class are used to define additional slots.

5.3.2. Adding Java objects to working memory

Once you've created an appropriate template, you can add some Java objects to working memory, automatically creating shadow facts to link them to Jess. Continuing with our `Account` example, imagine that you want to create an `Account`, and you want to add this object to working memory because rules will be working with it. Then all you need to do is:

```
Jess> (bind ?a (new Account))

<Java-Object:Account>

Jess> (add ?a)

<Fact-0>

Jess> (facts)

f-0   (MAIN::Account
             (balance 0.0)
             (class <Java-Object:java.lang.Class>)
             (OBJECT <Java-Object:Account>))
For a total of 1 facts in module MAIN.
```

The `add` function creates a shadow fact and links it to our `Account` object. We'll explore the nature of that link in the following section, but before we do that, I want to point out two things about `add` and about shadow fact templates in general.

First, note that the template has a slot named `OBJECT`. All shadow fact templates have this slot. Each shadow fact created from this template will use that slot to hold a reference to the Java object itself. Therefore, the original object is always easily available. The reverse mapping (given a Java object, finding its shadow fact) is also available using the method `jess.Rete.getShadowFactForObject(java.lang.Object)`.

Second, the `add` function understands the template naming convention discussed in the previous section. If you call `add` without creating a matching template first, `add` will create a template automatically, using that naming convention.

5.3.3. Responding to changes

Continuing with our `Account` example, we can use the `modify` function to change the fact, and the Java object will be automatically modified too:

```
Jess> (printout t (?a getBalance) crlf)

0.0

Jess> (modify 0 (balance 1))

<Fact-0>

Jess> (facts)

f-0   (MAIN::Account
                  (balance 1)
                  (class <Java-Object:java.lang.Class>)
                  (OBJECT <Java-Object:Account>))
For a total of 1 facts in module MAIN.

Jess> (printout t (?a getBalance) crlf)

1.0
```

But what happens if we modify the `Account` directly?

```
Jess> (printout t (?a getBalance) crlf)

1.0

Jess> (?a setBalance 2)

Jess> (printout t (?a getBalance) crlf)

2.0

Jess> (facts)

f-0   (MAIN::Account
             (balance 1)
             (class <Java-Object:java.lang.Class>)
             (OBJECT <Java-Object:Account>))
For a total of 1 facts in module MAIN.
```

The working memory still thinks our Account's balance is 1.0. There are several ways to notify Jess that the object is changed and working memory needs updating. One is using the `update` function:

```
Jess> (update ?a)

Jess> (facts)

f-0   (MAIN::Account
             (balance 2.0)
             (class <Java-Object:java.lang.Class>)
             (OBJECT <Java-Object:Account>))
For a total of 1 facts in module MAIN.
```

`update` tells Jess to refresh the slot values of the shadow fact linked to the given Java object. The `reset` function, which resets working memory, updates the slots of *all* shadow facts in the

process. You can also actively tell Jess to refresh its knowledge of the properties of individual objects using the `jess.Rete.updateObject(java.lang.Object)` method.

This behaviour is what you get for objects that don't support *property change notification.* In practice, this is fine. Most rule-based programs reason about static "value" objects whose properties don't change over time, or change only occasionally or at well-defined times.

But if your objects will be changed often from outside of Jess, then it would be nice to have updates happen automatically. If you want to have your shadow facts stay continuously up to date, Jess needs to be notified whenever a Bean property changes. For this to happen, the Bean has to support the use of `java.beans.PropertyChangeListener`s. For Beans that fulfill this requirement Jess will automatically arrange for working memory to be updated every time a property of the Bean changes. We can modify our Account class to support this feature like this:

```
import java.io.Serializable;
import java.beans.*;

import java.io.Serializable;

public class PCSAccount implements Serializable {
    private PropertyChangeSupport pcs = new PropertyChangeSupport(this);
    private float balance;
    public float getBalance() { return balance; }
    public void setBalance(float balance) {
        float temp = this.balance;
        this.balance = balance;
        pcs.firePropertyChange("balance", new Float(temp), new Float(balance));
    }
    public void addPropertyChangeListener(PropertyChangeListener pcl) {
        pcs.addPropertyChangeListener(pcl);
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl) {
        pcs.removePropertyChangeListener(pcl);
    }
    // Other, more interesting methods
}
```

Now whenever an Account balance is modified, Jess will be notified and the corresponding shadow fact will be updated immediately.

5.3.4. More about PropertyChangeEvents

PropertyChangeEvents and shadow facts seem magical to some people; the flow of control can be confusing, and the notion of things happening "automatically" sometimes makes people forget that no code executes in Java unless something, somewhere, invokes it. They're not magical, and to prove it, I'll explain exactly where they come from, what they do, and what Jess does when it sees one -- or not.

First, have a look back at the PCSAccount class above. It's a classic example of a JavaBean that supports PropertyChangeListeners. When you add an instance of this class to working memory, Jess will *usually* call `addPropertyChangeListener(listener)` on it, where *listener* is a Jess object designed to respond to change events. The listener object is added to a list of

objects that want to be notified when the PCSAccount object's properties change. This list is managed by the `PropertyChangeSupport` object that the PCSAccount holds as a member (Why "usually?" If you use `definstance` to add the object to working memory and specify `static` as the final argument, then Jess will skip this step.)

Of course, many (most) Java classes *don't* allow you to add PropertyChangeListeners to them, and so for many of the Java objects you'll add to working memory, Jess doesn't register itself with the object in any way. The object is filed away in working memory, and that's it.

There are now two main scenarios of interest: an object in working memory can be modified from Jess code, or from Java code. Each of these scenarios can happen with or without PropertyChangeListeners, for a total of four cases. We'll discuss each of these four situations separately. The most important things to remember is that all of the steps discussed here happen synchronously: neither Jess, nor the `PropertyChangeSupport` class, will spawn any new threads to process change events. Everything happens on the thread that initiates the change. Now, on to the four cases:

1. *Object modified from Java, no PropertyChangeListeners.* In this case, the Java code modifies the object, and Jess doesn't know about it. Bad things may happen as a result: Jess's working memory indices may need to be updated. Therefore you should always call the `jess.Rete.updateObject(java.lang.Object)` method to tell Jess when an object is changed in this way.
2. *Object modified from Java, PropertyChangeListeners.* In this case, Java code calls a method like `setBalance()` above, which calls `firePropertyChange()`, which iterates over that list of listeners we mentioned previously and calls `propertyChanged()` on each listener. One of those listeners will be the Jess object. Jess will examine the PropertyChangeEvent object that `propertyChange()` receives as an argument and update working memory appropriately.
3. *Object modified from Jess, no PropertyChangeListeners.* If you use `modify` (or the Java equivalent) to modify a shadow fact, then Jess will call the appropriate "setter" methods on the object, and update working memory.
4. *Object modified from Jess, PropertyChangeListeners.* This is the most complicated case. Note that there is a problem: if Jess calls a setter method on the object, then the object will send a redundant `PropertyChangeEvent` back to Jess. Unless Jess is careful, there could be an infinite loop. What Jess does is to call the setter methods, and then rely on the object to send a `PropertyChangeEvent` back. When Jess receives the change event, it will update working memory in response. This works fine as long as property change notification is implemented correctly by the object. If the object accepts `PropertyChangeListeners` but then doesn't send `PropertyChangeEvents`, however, then working memory will not be updated. If you have a class that behaves this way and you can't fix it, then you can always use `definstance` with the `static` option so that Jess ignores the change support.

5.3.5. Shadow fact miscellany

Shadow fact templates, like all templates, can `extend` one another. In fact, shadow fact templates can extend ordinary templates, and ordinary templates can extend shadow fact templates. Of course, for a shadow fact template to extend a plain template, the corresponding Java class must have property names that match the plain template's slot names. Note, also, that just because two Java classes have an inheritance relationship doesn't mean that

templates created from them will have the same relationship. You must explicitly declare all such relationships using `extends`. See the full documenation for `deftemplate` for details.

One final note about Java Beans used with Jess: Beans are often operating in a multithreaded environment, and so it's important to protect their data with synchronized blocks or synchronized methods. However, sending `PropertyChangeEvents` while holding a lock on the Bean itself can be dangerous, as the Java Beans Specification points out:

"In order to reduce the risk of deadlocks, we strongly recommend that event sources should avoid holding their own internal locks when they call event listener methods. Specifically, as in the example code in Section 6.5.1, we recommend they should avoid using a synchronized method to fire an event and should instead merely use a synchronized block to locate the target listeners and then call the event listeners from unsynchronized code." -- JavaBean Specification, v 1.0.1, p.31.
Failing to heed this advice can indeed cause deadlocks in Jess.

5.4. Ordered facts

Most of the time, you will use unordered facts (or their cousins, shadow facts.) They are nicely structured, and they're the most efficient kind of fact in Jess. In some cases, though, slot names are redundant, and force you to do more typing than you'd like. For example, if a fact represents a single number, it seems silly to use an unordered fact like this:

```
(number (value 6))
```

What you'd like would be a way to leave out that redundant "value" identifier. Ordered facts let you do exactly that.

Ordered facts are simply Jess lists, where the first field (the *head* of the list) acts as a sort of category for the fact. Here are some examples of ordered facts:

```
(shopping-list eggs milk bread)
(person "Bob Smith" Male 35)
(father-of danielle ejfried)
```

You can add ordered facts to the working memory using the `assert` function, just as with unordered facts. If you add a fact to working memory whose head hasn't been used before, Jess assumes you want it to be an ordered fact and creates an appropriate template automatically. Alternatively, you can explicitly declare an ordered template using the `ordered` declaration with the deftemplate construct:

```
Jess> (deftemplate father-of
            "A directed association between a father and a child."
            (declare (ordered TRUE)))
```

The quoted string is a *documentation comment*; you can use it to describe the template you're defining. Although declaring ordered templates this way is optional, it's good style to declare all your templates.

Note that an ordered fact is very similar to an unordered fact with only one multislot. The similarity is so strong, that in fact this is how ordered facts are implemented in Jess. If you assert an ordered fact, Jess automatically generates a template for it. This generated template will contain a single slot named "__data". Jess treats these facts specially - the name of the slot is normally hidden when the facts are displayed. This is really just a syntactic shorthand, though; ordered facts really are just unordered facts with a single multislot named "__data".

5.5. The deffacts construct

Typing separate `assert` commands for each of many facts is rather tedious. To make life easier in this regard, Jess includes the `deffacts` construct. A `deffacts` construct is a simply a named list of facts. The facts in all defined `deffacts` are asserted into the working memory whenever a `reset` command is issued:

```
Jess> (deffacts my-facts "Some useless facts"
  (foo bar)
  (bar foo))

TRUE

Jess> (reset)

TRUE

Jess> (facts)

f-0   (MAIN::initial-fact)
f-1   (MAIN::foo bar)
f-2   (MAIN::bar foo)
For a total of 3 facts in module MAIN.
```

5.6. How Facts are Implemented

Every fact, shadow or otherwise, corresponds to a single instance of the `jess.Fact` class. You can learn more about this class here. Templates are represented by instances of `jess.Deftemplate`, which you can read about here.

5. Working Memory

6. Making Your Own Rules

6.1. Introducing defrules

Now that we've learned how to populate Jess's working memory, we can answer the obvious question: what is it good for? The answer is that `defquery`s can search it to find relationships between facts, and `defrule`s can take actions based on the contents of one or more facts.
A Jess rule is something like an `if... then` statement in a procedural language, but it is not used in a procedural way. While `if... then` statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their `if` parts (their *left-hand-sides* or *LHS*s) are satisfied, given only that the rule engine is running. This makes Jess rules less deterministic than a typical procedural program. See the chapter on the Rete algorithm for an explanation of why this architecture can be many orders of magnitude faster than an equivalent set of traditional `if... then` statements.
In this chapter we're going to make a lot of use of a "person" template that looks like this:

```
Jess> (deftemplate person (slot firstName) (slot lastName) (slot age))
```

Rules are defined in Jess using the `defrule` construct. A very simple rule looks like this:

```
Jess> (defrule welcome-toddlers
    "Give a special greeting to young children"
    (person {age < 3})
    =>
    (printout t "Hello, little one!" crlf))
```

This rule has two parts, separated by the "=>" symbol (which you can read as "then".) The first part consists of the LHS *pattern* `(person {age < 3})`. The second part consists of the RHS *action*, the call to `println`. If you're new to Jess, it can be hard to tell the difference due to the LISP-like syntax, but the LHS of a rule consists of patterns which are used to match facts in the working memory, while the RHS contains function calls.
*The LHS of a rule (the "if" part) consists of patterns that match facts, **NOT** function calls. The actions of a rule (the "then" clause) are made up of function calls. The following rule does **NOT** work:*

```
Jess> (defrule wrong-rule
  (eq 1 1)
  =>
  (printout t "Just as I thought,  1 == 1!" crlf))
```

*This rule will NOT fire just because the function call `(eq 1 1)` would evaluate to true. Instead, Jess will try to find a fact in the working memory that looks like `(eq 1 1)`. Unless you have previously asserted such a fact, this rule will **NOT** be activated and will not fire. If you want to fire a rule based on the evaluation of a function that is not related to a pattern, you can use the test CE.*
Our example rule, then, will be activated when an appropriate `(person)` fact appears in the working memory. When the rule executes, or *fires*, a message is printed. Let's turn this rule into a complete program. The function `watch all` tells Jess to print some useful diagnostics as we enter our program.

```
Jess> (deftemplate person (slot firstName) (slot lastName) (slot age))

TRUE

Jess> (watch all)
```

```
TRUE

Jess>  (reset)

==> f-0 (MAIN::initial-fact)
TRUE

Jess> (defrule welcome-toddlers
        "Give a special greeting to young children"
        (person {age < 3})
        =>
        (printout t "Hello, little one!" crlf))

welcome-toddlers: +1+1+1+t
TRUE

Jess>  (assert (person (age 2)))

==> f-1 (MAIN::person (firstName nil) (lastName nil) (age 2))
==> Activation: MAIN::welcome-toddlers :  f-1
<Fact-1>
```

Some of these diagnostics are interesting. We see first of all how issuing the `reset` command asserts the fact `(initial-fact)`. You should always issue a `reset` command when working with rules. When the rule itself is entered, we see the line "+1+1+t". This tells you something about how the rule is interpreted by Jess internally (see The Rete Algorithm for more information.) When the fact `(person (age 2))` is asserted, we see the diagnostic "Activation: MAIN::welcome-toddlers : f-1". This means that Jess has noticed that the rule `welcome-toddlers` has all of its LHS conditions met by the given list of facts ("f-1").

After all this, our rule didn't fire; why not? Jess rules only fire while the rule engine is running (although they can be *activated* while the engine is not running.) To start the engine running, we issue the `run` command.

```
Jess> (run)

FIRE 1 MAIN::welcome-toddlers f-1
Hello, little one!
<== Focus MAIN
1
```

As soon as we enter the `run` command, the activated rule fires. Since we have `watch all`, Jess prints the diagnostic `FIRE 1 welcome-toddlers f-1` to notify us of this. We then see the output of the rule's RHS actions. The final number "1" is the number of rules that fired (it is the return value of the `run` command.) The `run` function returns when there are no more activated rules to fire.

What would happen if we entered `(run)` again? Nothing. A rule will be activated only once for a given set of facts; once it has fired, that rule will not fire again for the same list of facts. We won't print the message again until another toddler shows up.

Rules are uniquely identified by their name. If a rule named `my-rule` exists, and you define another rule named `my-rule`, the first version is deleted and will not fire again, even if it was activated at the time the new version was defined.

6.2. Simple patterns

A pattern is always a set of parentheses including the name of the fact to be matched plus zero or more slot descriptions. There are now two kinds of slot descriptions in Jess: the new-style

"simplified" or "Java" syntax, and the old-style, more complex but more powerful syntax. New-style slot descriptions are enclosed in curly braces, like the one in our `welcome-toddlers` rule:

```
Jess> (defrule welcome-toddlers
    "Give a special greeting to young children"
    (person {age < 3})
    =>
    ((System.out) println "Hello, little one!"))
```

Old-style slot descriptions use parentheses instead of curly braces. The syntax allowed for the two kinds of descriptions are different. We'll talk mostly about the new simplified syntax in this section, and save most of the old-style syntax for the next section. However, there is one very easy and very important thing you can do with the old-style syntax that we'll need right away: you can declare a variable to refer to the contents of a slot. For example, look at the following pattern:

```
(person (age ?a) (firstName ?f) (lastName ?l))
```

This pattern will match any person fact. When the rule fires, Jess will assign the contents of the "age" slot to a variable "?a", the firstName slot to a variable "?f", and the lastName slot to "?l". You'll be able to use these variables elsewhere in the same rule, both on the left-hand side and on the right-hand side.

The simplified slot descriptions we'll talk about in this section are Java-like Boolean expressions (infix expressions) using the following operators:

- < (less than)
- <= (less than or equal to)
- > (greater than)
- <= (greater than or equal to)
- == (equals)
- != (not equal to)
- <> (not equal to, alternate syntax)
- && (and)
- || (or)

These operators are used in infix format, like Java operators. Within a simplified pattern, a symbol stands for the value of the slot in that pattern with the same name. The syntax is simple and a few examples will serve to document it. The first example matches any person who is between the ages of 13 and 19, inclusive:

```
Jess> (defrule teenager
    ?p <- (person {age > 12 && age < 20} (firstName ?name))
    =>
    (printout t ?name " is " ?p.age " years old." crlf))
```

The variable "?p" is a *pattern binding*; it's bound to the whole fact that matches this pattern. Note how we use the dotted variable syntax to get the the value of the "age" slot. We declared a variable for the `firstName` slot in an old-style slot description.

You can write tests that look at several slots at once:

```
Jess> (defrule same-first-and-last-name
    (person {firstName == lastName})
    =>
    (printout t "That is a funny name!" crlf))
```

You can use parentheses to group operations in Java patterns; for example:

```
Jess> (defrule teenage-or-bob
    (person {(age > 12 && age < 20) || firstName == Bob})
    =>
    (printout t "The person is a teenager, or is named 'Bob'." crlf))
```

6.2.1. Using multiple simple patterns together

Most rules have more than one pattern -- often many more. What's more, patterns relate to one another. For example, we might want to find two unrelated people who are the same age. To do this sort of task, you simply match one person, then match a second person and compare them to the first. To compare the two facts, we need to bind variables to them so we can refer to them; then we use a special "dot notation" to refer to the slots of the first fact:

```
Jess> (defrule two-same-age-different-name
    ?person1 <- (person)
    ?person2 <- (person {age == person1.age &&lastName != person1.lastName})
    =>(printout t "Found two different " ?person1.age "-year-old people." crlf))
```

The variable "?person1" is another *pattern binding*. Note that when you refer to a pattern binding in a Java pattern, the "?" variable indicator is omitted.

6.3. Patterns in Depth

The curly-brace notation we looked at in the previous section is a simplified way of writing patterns that fills many basic needs. But Jess actually supports a richer syntax that gives you more capabilities. One limitation of the curly-brace notation is that it can only be used with unordered facts. It is this richer syntax that we'll cover here. Whereas the simplified slot patterns use curly braces, the richer syntax uses parentheses to enclose slots.

As previously shown, you can specify a variable name for a field in any of a rule's patterns (but not the pattern's head). A variable matches any value in that position within a rule. For example, the rule:

```
Jess> (deftemplate coordinate (slot x) (slot y))

Jess> (defrule example-2
```

```
    (coordinate (x ?x) (y ?y))
    =>
    (printout t "Saw 'coordinate " ?x " " ?y "'" crlf))
```

will be activated once for every `coordinate` fact. The variables ?x and ?y matched in the pattern are available in the actions on the RHS of the same rule.

A slot descriptor can also include any number of tests to qualify what it will match. Tests follow the variable name and are separated from it and from each other by an *and* (&) or *or* (|) symbol. (The variable name itself is actually optional.) Tests can be:

- A literal value (in which case the variable matches *only* that value); for example, the value `1.0` in `(coordinate (x 1.0))`.
- A variable which was assigned earlier on the rule's LHS. This will constrain the field to contain the same value as the variable was first bound to; for example, `(coordinate (x ?x) (y ?x))` will only match coordinates facts with equal x and y values.
- A colon (`:`) followed by a function call, in which case the test succeeds if the function returns the special value `TRUE`. These are called *predicate constraints*; for example, `(cordinate (x ?x&:(> ?x 10)))` matches "coordinate" facts with x greater 10. There is a powerful shortcut way to write many predicate constraints which we'll look at in a minute.
- An equals sign (`=`) followed by a function call. In this case the field must match the return value of the function call. These are called *return value constraints*. Note that both predicate constraints and return-value constraints can refer to variables bound elsewhere in this or any preceding pattern in the same rule. ***Note****:* pretty-printing a rule containing a return value contstraint will show that it has been transformed into an equivalent predicate constraint. An example of a return-value constraint would be

```
(coordinate (x ?x) (y =(+ ?x 1)))
```

  which matches coordinates with y one greater than x.

- A Java regular expression surrounded by "/" characters. The field must match the given regular expression (regular expressions are available only when you're using Jess under JDK 1.4 and up.) For example, the pattern `(person (name /A.*/)` matches people whose first initial is "A".
- Any of the other options preceded by a tilde (~), in which case the sense of the test is reversed (inequality or false); for example `(coordinate (x ?x) (y ~?x))` matches coordinates in which x and y differ.

Ampersands (&) represent logical "and", while pipes (|) represent logical "or." & has a higher precedence than |, so that the following

```
(foo ?X&:(oddp ?X)&:(< ?X 100)|0)
```

matches a `foo` fact with a single field containing either an odd number less than 100, or 0. Here's an example of a rule that uses several kinds of tests:

```
Jess> (defrule example-3
  (not-b-and-c ?n1&~b ?n2&~c)
  (different ?d1 ~?d1)
  (same ?s ?s)
  (more-than-one-hundred ?m&:(> ?m 100))
  (red-or-blue red|blue)
  =>
```

6. Making Your Own Rules

```
  (printout t "Found what I wanted!" crlf))
```

The first pattern will match an ordered fact with head `not-b-and-c` with exactly two fields such that the first is not `b` and the second is not `c`. The second pattern will match any fact with head `different` and two fields such that the two fields have different values. The third pattern will match a fact with head `same` and two fields with identical values. The fourth pattern matches a fact with head `more-than-one-hundred` and a single field with a numeric value greater than 100. The last pattern matches a fact with head `red-or-blue` followed by either the symbol `red` or the symbol `blue`.

If you match to a defglobal with a pattern like (foo ?*x*), the match will only consider the value of the defglobal when the fact is asserted. Subsequent changes to the defglobal's value will *not* invalidate the match - i.e., the match does not reflect the current value of the defglobal, but only the value at the time the matching fact was asserted.

6.4. Matching in Multislots

Pattern matching in multislots (and in ordered facts, which are really just facts with a single multislot whose name is hidden) is similar to matching in regular slots. The main difference is that you may include separate clusters of tests for each *field* within a multislot. The number of clusters implicitly specifies the number of items in a matching multislot. So, for example, the `grocery-list` pattern in the following rule matches only grocery lists with exactly three items:

```
Jess> (defrule match-three-items
  (grocery-list ? ? ?)
  =>
  (printout t "Found a three-item list" crlf))

TRUE

Jess> (assert (grocery-list eggs milk bacon))

<Fact-0>

Jess> (run)

Found a three-item list
1
```

Note that, as shown here, you can match a field without binding it to a named variable by omitting the variable name and using just a question mark (`?`) as a placeholder.

You can match any number (zero or more) of fields in a multislot or ordered fact using a *multifield*. A multifield is just a variable constraint preceded by a '$' character. The matched items are used to construct a list, and the list is assigned to that variable:

```
Jess> (defrule match-whole-list
  (grocery-list $?list)
  =>
  (printout t "I need to buy " ?list crlf))

TRUE
```

```
Jess> (assert (grocery-list eggs milk bacon))

<Fact-0>

Jess> (run)

I need to buy (eggs milk bacon)
1
```

Multifields can be used in combination with other kinds of tests, and they're a very convenient way of saying "... and some other stuff." For example, this rule matches grocery lists containing bacon in any position. It does this by using two blank multifields: one to match all the items before bacon in the list, and the other (which in this case, will match zero items) to match all the items after.

```
Jess> (defrule match-list-with-bacon
  (grocery-list $? bacon $?)
  =>
  (printout t "Yes, bacon is on the list" crlf))

TRUE

Jess> (assert (grocery-list eggs milk bacon))

<Fact-0>

Jess> (run)

Yes, bacon is on the list
1
```

Finally, note that a multifield is not a special kind of variable. When a multifield $?list is matched, it's the variable ?list that receives the value.

6.5. Pattern bindings

Sometimes you need a handle to an actual fact that helped to activate a rule. For example, when the rule fires, you may need to retract or modify the fact. To do this, you use a pattern-binding variable:

```
Jess> (defrule example-5
  ?fact <- (a "retract me")
  =>
  (retract ?fact))
```

The variable (`?fact`, in this case) is bound to the particular fact that activated the rule. Note that ?fact is a jess.Value object of type RU.FACT, not an integer. It is basically a reference to a jess.Fact object. You can convert an ordinary number into a FACT using the `fact-id` function. You can convert a FACT into an integer when necessary by using reflection to call the `Fact.getFactId()` function. The `jess.Value.factValue()` method can be called on a FACT Value to obtain the actual `jess.Fact` object from Java code. In Jess code, a fact-id essentially *is* a `jess.Fact`, and you can call `jess.Fact` methods on a fact-id directly:

```
Jess> (defrule example-5-1
  ?fact <- (initial-fact)
  =>
```

```
  (printout t (call ?fact getName) crlf))

TRUE

Jess> (reset)

TRUE

Jess> (run)

initial-fact
1
```

See the section on the jess.FactIDValue class for more information.

Note that once a fact is asserted, Jess will always use the same `jess.Fact` object to represent it, even if the original fact is modified. Therefore, you can store references to fact objects in the slots of other facts as a way of representing structured data.

6.6. More about regular expressions

Jess's new regular expression facility builds on Java's `java.util.regex` package. This section presents a few examples of how it works.

```
Jess> (defrule rule-1
  (foo /xy+z/)
  =>)

(defrule rule-2
  (foo a ?x&/d*ef/)
  (bar ?x)
  =>)
```

The first rule matches a "foo" fact with a single field of the form "xyz", "xyyz", "xyyyz"... The second rule matches a "foo" fact with two fields: the symbol "a" followed by a string or symbol of the form "ef", "def", "ddef"...; this string is bound to the variable ?x and matched to the only field in the second pattern. Patterns are always matched against the entire contents of the field. You can write `/.*abc.*` to match for an embedded string "abc".

There is also a function `regexp` which can be used in procedural code. In this release, it just takes two arguments, a regular expression and a target string, and returns a boolean result.

6.7. Salience and conflict resolution

Each rule has a property called *salience* that is a kind of rule priority. Activated rules of the highest salience will fire first, followed by rules of lower salience. To force certain rules to always fire first or last, rules can include a salience declaration:

```
Jess> (defrule example-6
  (declare (salience -100))
  (command exit-when-idle)
  =>
  (printout t "exiting..." crlf))
```

Declaring a low salience value for a rule makes it fire after all other rules of higher salience. A high value makes a rule fire before all rules of lower salience. The default salience value is zero.

Salience values can be integers, global variables, or function calls. See the `set-salience-evaluation` command for details about when such function calls will be evaluated.
The order in which multiple rules of the same salience are fired is determined by the active *conflict resolution strategy*. Jess comes with two strategies: "depth" (the default) and "breadth." In the "depth" strategy, the most recently activated rules will fire before others of the same salience. In the "breadth" strategy, rules fire in the order in which they are activated. In many situations, the difference does not matter, but for some problems the conflict resolution strategy is important. You can write your own strategies in Java; see the chapter on extending Jess with Java for details. You can set the current strategy with the `set-strategy` command.
Note that the use of salience is generally discouraged, for two reasons: first it is considered bad style in rule-based programming to try to force rules to fire in a particular order. Secondly, use of salience will have a negative impact on performance, at least with the built-in conflict resolution strategies.
You can see the list of activated, but not yet fired, rules with the `agenda` command.

6.8. The 'and' conditional element.

Any number of patterns can be enclosed in a list with `and` as the head. The resulting pattern is matched if and only if all of the enclosed patterns are matched. By themselves, `and` groups aren't very interesting, but combined with or and not conditional elements, they can be used to construct complex logical conditions.
The entire left hand side of every rule and query is implicitly enclosed in an `and` conditional element.

6.9. The 'or' conditional element.

Any number of patterns can be enclosed in a list with `or` as the head. The resulting pattern is matched if one or more of the patterns inside the `or` are matched. If more than one of the subpatterns are matched, the `or` is matched more than once:

```
Jess> (defrule or-example-1
   (or (a) (b) (c))
   =>)

Jess> (assert (a) (b) (c))

Jess> (printout t (run) crlf)

3
```

An `and` group can be used inside of an `or` group, and vice versa. In the latter case, Jess will rearrange the patterns so that there is a single `or` at the top level. For example, the rule

```
Jess> (defrule or-example-2a
    (and (or (a)
             (b))
         (c))
      =>)
```

will be automatically rearranged to

```
Jess> (defrule or-example-2b
    (or (and (a) (c))
        (and (b) (c)))
    =>)
```

DeMorgan's second rule of logical equivalence, namely

```
(not (or (x) (y)))  => (and (not (x)) (not (y)))
```
will be used when necessary to hoist an `or` up to the top level.
Note that if the right hand side of a rule uses a variable defined by matching on the left hand side of that rule, and the variable is defined by one or more branches of an `or` pattern but not all branches, then a runtime error may occur.

6.10. The 'not' conditional element.

Any single pattern can be enclosed in a list with `not` as the head. In this case, the pattern is considered to match if a fact (or set of facts) which matches the pattern is *not* found. For example:

```
Jess> (defrule example-7
   (person ?x)
   (not (married ?x))
   =>
   (printout t ?x " is not married!" crlf))
```
Note that a `not` pattern cannot define any variables that are used in subsequent patterns (since a `not` pattern does not match any facts, it cannot be used to define the values of any variables!) You can introduce variables in a `not` pattern, so long as they are used only within that pattern; i.e,

```
Jess> (defrule no-odd-numbers
   (not (number ?n&:(oddp ?n)))
   =>
   (printout t "There are no odd numbers." crlf))
```
Similarly, a `not` pattern can't have a pattern binding.
A `not` CE is evaluated only when either a fact matching it exists, or when the pattern immediately before the `not` on the rule's LHS is evaluated. If a `not` CE is the first pattern on a rule's LHS, or is the the first the pattern in an `and` group, or is the only pattern on a given branch of an `or` group, the pattern `(initial-fact)` is inserted to become this important preceding pattern. Therefore, the fact `(initial-fact)` created by the `reset` command is necessary to the proper functioning of some `not` patterns. For this reason, it is especially important to issue a `reset` command before attempting to run the rule engine when working with `not` patterns.
Multiple `not` CEs can be nested to produce some interesting effects (see the discussion of the `exists` CE).
The `not` CE can be used in arbitrary combination with the and and or CEs. You can define complex logical structures this way. For example, suppose you want a rule to fire once if for every fact (a ?x), there is a fact (b ?x). You could express that as

```
Jess> (defrule forall-example
  (not (and (a ?x) (not (b ?x))))
  =>)
```
i.e., "It is not true that for some ?x, there is an (a ?x) and no (b ?x)". This is actually how the the `forall` CE is implemented.

6.11. The 'exists' conditional element.

A pattern can be enclosed in a list with `exists` as the head. An `exists` CE is true if there exist any facts that match the pattern, and false otherwise. `exists` is useful when you want a rule to fire only once, although there may be many facts that could potentially activate it.

```
Jess> (defrule exists-demo
    (exists (honest ?))
    =>
    (printout t "There is at least one honest man!" crlf))
```
If there are any honest men in the world, the rule will fire once and only once.

`exists` may not be combined in the same pattern with a `test` CE.

Note that `exists` is precisely equivalent to (and in fact, is implemented as) two nested `not` CEs; i.e., `(exists (A))` is the same as `(not (not (A)))`. It is rather common for people to write something like "`(not (exists (A)))`," but this is just a very inefficient way to write `(not (A))`.

6.12. The 'test' conditional element.

A pattern with `test` as the head is special; the body consists not of a pattern to match against the working memory but of a Boolean function. The result of evaluating this function determines whether the pattern matches. A `test` pattern fails if and only if the function evaluates to the symbol `FALSE`; if it evaluates to `TRUE` or any other value, the pattern with "match." For example:

```
Jess> (deftemplate person (slot age))

Jess> (defrule example-8
   (test (eq 4 (+ 2 2)))
   =>
   (printout t "2 + 2 is 4!" crlf))
```
Note that a `test` pattern, like a `not`, cannot define any variables for use in later patterns. `test` and `not` may be combined:
```
  (not (test (eq ?X 3)))
```
is equivalent to:
```
  (test (neq ?X 3))
```
A `test` CE is evaluated every time the *preceding* pattern on the rule's LHS is evaluated. Therefore the following two rules are precisely equivalent in behaviour:

```
Jess> (defrule rule_1
  (foo ?X)
  (test (> ?X 3))
  =>)

Jess> (defrule rule_2
  (foo ?X&:(> ?X 3))
  =>)
```

In fact, starting with Jess 7.1, the functions in a `test` CE are simply added to the previous pattern's tests. Therefore these are not only equivalent, but they are in fact identical in every respect. You can use `test` CEs wherever you'd like without worrying about performance implications. When you need to evaluate a complicated function during pattern matching, a `test` CE is often clearer than the equivalent slot test.

You should now understand why, for rules in which a `test` CE is the first pattern on the LHS or the first pattern in a branch of an `or` CE, the pattern `(initial-fact)` is inserted to become the "preceding pattern" for the `test`. The fact `(initial-fact)` is therefore also important for the proper functioning of the `test` conditional element; the caution about `reset` in the preceding section applies equally to `test`.

6.12.1. Time-varying method returns

One useful property of the `test` CE is that it's the only valid place to put tests whose results might change without the contents of any slot changing. For example, imagine that you've got two Java classes, `A` and `B`, and that `A` has a method `contains` which takes a `B` as an argument and returns boolean. Further, imagine that for any given `B` object, the return value of `contains` will change over time. Finally, imagine that you've defined shadow fact templates for both these classes and are writing rules to work with them. Under these circumstances, a set of patterns like this:

```
(A (OBJECT ?a))
(B (OBJECT ?b&:(?a contains ?b)))
```

is incorrect. If the return value of `contains` changes, the match will be invalidated and Jess's internal data structures may be corrupted. In particular, this kind of construct tends to cause memory leaks.
The correct way to express this same set of patterns is to use the `test` conditional element, like this:

```
(A (OBJECT ?a))
(B (OBJECT ?b))
(test (?a contains ?b))
```

The function `contains` is now guaranteed to be called at most once for each combination of target and argument, and so any variation in return value will have no impact.

6.12.2. When should I use test?

The `test` conditional element can be used whenever writing a test directly in a slot would be unclear. It is also useful with time-varying return values as described previously. There is no longer any performance penalty associated with the `test` conditional element.

6.13. The 'logical' conditional element.

The `logical` conditional element lets you specify *logical dependencies* among facts. All the facts asserted on the RHS of a rule become dependent on the matches to the `logical` patterns on that rule's LHS. If any of the matches later become invalid, the dependent facts are retracted automatically. In this simple example, a single fact is made to depend on another single fact:

```
Jess> (defrule rule-1
  (logical (faucet-open))
  =>
  (assert (water-flowing)))

TRUE

Jess> (assert (faucet-open))

<Fact-0>

Jess> (run)

1

Jess> (facts)

f-0   (MAIN::faucet-open)
```

```
f-1   (MAIN::water-flowing)
For a total of 2 facts in module MAIN.

Jess> (watch facts)

TRUE

Jess> (retract (fact-id 0))

<== f-0 (MAIN::faucet-open)
<== f-1 (MAIN::water-flowing)
TRUE
```

The `(water-flowing)` fact is logically dependent on the `(faucet-open)` fact, so when the latter is retracted, the former is removed, too.

A fact may receive logical support from multiple sources -- i.e., it may be asserted multiple times with a different set of logical supports each time. Such a fact isn't automatically retracted unless each of its logical supports is removed.

If a fact is asserted without explicit logical support, it is said to be *unconditionally supported.* If an unconditionally supported fact also receives explicit logical support, removing that support will not cause the fact to be retracted.

If one or more `logical` CEs appear in a rule, they must be the first patterns in that rule; i.e., a `logical` CE cannot be preceded in a rule by any other kind of CE.

Shadow facts are no different than other facts with regard to the `logical` CE. Shadow facts can provide logical support and can receive logical support. In the current implementation, shadow facts can only provide logical support as a whole. In a future version of Jess, it will be possible for a shadow fact to provide logical support based on any combination of individual slot values.

The `logical` CE can be used together with all the other CEs, including `not` and `exists`. A fact can thus be logically dependent on the non-existence of another fact, or on the existence of some category of facts in general.

The Jess language functions `dependents` and `dependencies` let you query the logical dependencies among facts.

6.14. The 'forall' conditional element.

The "forall" grouping CE matches if, for every match of the first pattern inside it, all the subsequent patterns match. An example:

```
Jess> (defrule every-employee-has-a-stapler-and-holepunch
  (forall (employee (name ?n))
          (stapler (owner ?n))
          (holepunch (owner ?n)))
   =>
  (printout t "Every employee has a stapler and a holepunch." crlf))
```

This rule fires if there are a hundred employees and everyone owns the appropriate supplies. If a single employee doesn't own the supplies, the rule won't fire.

6.15. The 'accumulate' conditional element.

The "`accumulate`" CE is complicated, and perhaps hard to understand, but it's incredibly powerful. It lets you count facts, add up fields, store data into collections, etc. I will eventually need to write quite a bit of documentation for it, but for now, the following should get you started.

The accumulate CE looks like this:

```
(accumulate <initializer> <action> <result> <conditional element>)
```

When an accumulate CE is encountered during matching (i.e., when the preceding pattern is matched, or when the contained CE is matched), the following steps occur:

1. A new execution context is created.
2. The initializer is executed in that context.
3. If the CE is activated via the left input, all the matching tokens from the right memory are considered. If it's activated via the right input, each of the matching left tokens are visited. As each is visited, all of its matching right tokens are considered in turn.
4. For each token considered, the variables it defines are bound in the execution context, and the action is executed.
5. If a pattern binding is present, the result is bound to the given variable.
6. Finally, the accumulate CE matches successfully and matching continues at the next conditional element.

What this all means is that "accumulate" lets you execute some code for every match, and returns the accumulated result. For example, this rule counts the number of employees making more than $100,000 per year. A variable is initialized to zero, and incremented for every match; that variable is then bound to the pattern binding.

```
Jess> (deftemplate employee (slot salary) (slot name))

Jess> (defrule count-highly-paid-employees
  ?c <- (accumulate (bind ?count 0)                       ;; initializer
                    (bind ?count (+ ?count 1))               ;; action
                    ?count                                   ;; result
                    (employee (salary ?s&:(> ?s 100000)))) ;; CE
  =>
  (printout t ?c " employees make more than $100000/year." crlf))
```

This variation prints a list of those employees instead by storing all the names in an ArrayList:

```
Jess> (defrule count-highly-paid-employees
  ?c <- (accumulate (bind ?list (new java.util.ArrayList))   ;; initializer
                    (?list add ?name)                         ;; action
                     ?list                                    ;; result
                    (employee (name ?name)
                              (salary ?s&:(> ?s 100000))))  ;; CE
  =>
  (printout t (?c toString)  crlf))
```

Warning: note that because matching one fact can cause `accumulate` to iterate over a large number of other facts, it can be computationally expensive. Do think about what you're doing when you use it.

Finally, note that `accumulate` is non-reentrant. You cannot nest one `accumulate` CE inside another, directly or indirectly.

6. Making Your Own Rules

6.16. The 'unique' conditional element.

The `unique` CE has been removed. The parser will accept but ignore it.

6.17. Node index hash value.

The *node index hash value* is a tunable performance-related parameter that can be set globally or on a per-rule basis. A small value will save memory, possibly at the expense of performance; a larger value will use more memory but lead to faster rule LHS execution.
In general, you might want to declare a large value for a rule that was likely to generate many partial matches (prime numbers are the best choices:)

```
Jess> (defrule nihv-demo
  (declare (node-index-hash 169))
  (item ?a)
  (item ?b)
  (item ?c)
  (item ?d)
  =>)
```

See the discussion of the `set-node-index-hash` function for a full discussion of this value and what it means.

6.18. The 'slot-specific' declaration for deftemplates

Deftemplate definitions can now include a "declare" section just as defrules can. There are several different properties that can be declared. One is "slot-specific". A template with this declaration will be matched in a special way: if a fact, created from such a template, which matches the left-hand-side of a rule is modified, the result depends on whether the modified slot is named in the pattern used to match the fact. As an example, consider the following:

```
Jess> (deftemplate D (declare (slot-specific TRUE)) (slot A) (slot B))

Jess> (defrule R
  ?d <- (D (A 1))
  =>
  (modify ?d (B 3)))
```

Without the "slot-specific" declaration, this rule would enter an endless loop, because it modifies a fact matched on the LHS in such a way that the modified fact will still match. With the declaration, it can simply fire once. This behavior is actually what many new users expect as the default; the technical term for it is *refraction.*

6.19. The 'no-loop' declaration for rules

If a rule includes the declaration `(declare (no-loop TRUE))`, then nothing that a rule does while firing can cause the immediate reactivation of the same rule; i.e., if a no-loop rule matches a fact, and the rule modifies that same fact such that the fact still matches, the rule will not be put back on the agenda, avoiding an infinite loop. This is basically just a stronger form of "slot-specific."

6.20. Removing rules

You can undefine a rule with the `jess.Rete.removeDefrule(String)` method. This will remove the rule completely from the engine.

6.21. Forward and backward chaining

The rules we've seen so far have been *forward-chaining* rules, which basically means that the rules are treated as `if... then` statements, with the engine passively executing the RHSs of activated rules. Some rule-based systems, notable Prolog and its derivatives, support *backward chaining.* In a backwards chaining system, rules are still `if... then` statements, but the engine seeks steps to activate rules whose preconditions are not met. This behaviour is often called "goal seeking". Jess supports both forward and backward chaining. Note that the explanation of backward chaining in Jess is necessarily simplified here since full explanation requires a good understanding of the underlying algorithms used by Jess.
To use backward chaining in Jess, you must first declare that certain fact templates will be *backward chaining reactive*. You can do this when you define the template:

```
Jess> (deftemplate factorial
  (declare (ordered TRUE)
  (backchain-reactive TRUE))
```

Alternatively, you can use the `do-backward-chaining` function after the template is defined:

```
Jess> (do-backward-chaining factorial)
```

Then you can define rules which match such patterns. Note that templates must be declared to be backwards chaining reactive *before* you define any rules which use the template.

```
Jess> (defrule print-factorial-10
  (factorial 10 ?r1)
  =>
  (printout t "The factorial of 10 is " ?r1 crlf))
```

When the rule compiler sees that a pattern matches a backward chaining reactive template, it rewrites the rule and inserts some special code into the internal representation of the rule's LHS. This code asserts a fact onto the fact-list that looks like

```
(need-factorial 10 nil)
```

if, when the rule engine is reset, there are no matches for this pattern. The head of the fact is constructed by taking the head of the reactive pattern and adding the prefix "need-".
Now, you can write rules which match these need-(x) facts.

```
Jess> (defrule do-factorial
  (need-factorial ?x ?)
  =>
  (bind ?r 1)
  (bind ?n ?x)
  (while (> ?n 1)
    (bind ?r (* ?r ?n))
    (bind ?n (- ?n 1)))
  (assert (factorial ?x ?r)))
```

The rule compiler rewrites rules like this too: it adds a negated match for the factorial pattern itself to the rule's LHS.
The end result is that you can write rules which match on (factorial), and if they are close to firing except they need a (factorial) fact to do so, any (need-factorial) rules may be activated. If these rules fire, then the needed facts appear, and the (factorial)-matching rules fire. This, then, is backwards chaining! Jess will chain backwards through any number of reactive patterns. For example:

```
Jess> (do-backward-chaining foo)

TRUE

Jess> (do-backward-chaining bar)

TRUE

Jess> (defrule rule-1
  (foo ?A ?B)
  =>
  (printout t foo crlf))

TRUE

Jess> (defrule create-foo
  (need-foo $?)
  (bar ?X ?Y)
  =>
  (assert (foo A B)))

TRUE

Jess> (defrule create-bar
  (need-bar $?)
  =>
  (assert (bar C D)))

TRUE

Jess> (reset)

TRUE

Jess> (run)

foo
3
```

In this example, none of the rules can be activated at first. Jess sees that `rule-1` could be activated if there were an appropriate `foo` fact, so it generates the request `(need-foo nil nil)`. This matches part of the LHS of rule `create-foo` cannot fire for want of a `bar` fact. Jess therefore creates a `(need-bar nil nil)` request. This matches the LHS of the rule `create-bar`,which fires and asserts `(bar C D)`. This activates `create-foo,` which fires, asserts `(foo A B)`, thereby activating `rule-1`, which then fires.

There is a special conditional element, `(explicit),` which you can wrap around a pattern to inhibit backwards chaining on an otherwise reactive pattern.

6.22. Defmodules

A typical rule-based system can easily include hundreds of rules, and a large one can contain many thousands. Developing such a complex system can be a difficult task, and preventing such a multitude of rules from interfering with one another can be hard too.

You might hope to mitigate the problem by partitioning a rule base into manageable chunks. *Modules* let you divide rules and templates into distinct groups. The commands for listing constructs let you specify the name of a module, and can then operate on one module at a time. If you don't explicitly specify a module, these commands (and others) operate by default on the *current module.* If you don't explicitly define any modules, the current module is always the *main*

module, which is named MAIN. All the constructs you've seen so far have been defined in MAIN, and therefore are often preceded by "MAIN::" when displayed by Jess.
Besides helping you to manage large numbers of rules, modules also provide a control mechanism: the rules in a module will fire only when that module has the *focus*, and only one module can be in focus at a time.
Note for CLIPS users: Jess's `defmodule` construct is similar to the CLIPS construct by the same name, but it is not identical. The syntax and the name resolution mechanism are simplified. The focus mechanism is much the same.

6.22.1. Defining constructs in modules

You can define a new module using the `defmodule` construct:

```
Jess> (defmodule WORK)

TRUE
```

You can place a deftemplate, defrule, or deffacts into a specific module by qualifying the name of the construct with the module name:

```
Jess> (deftemplate WORK::job (slot salary))

TRUE

Jess> (list-deftemplates WORK)

WORK::job
For a total of 1 deftemplates in module WORK.
```

Once you have defined a module, it becomes the *current module:*

```
Jess> (get-current-module)

MAIN

Jess> (defmodule COMMUTE)

TRUE

Jess> (get-current-module)

COMMUTE
```

If you don't specify a module, all deffacts, templates and rules you define will automatically become part of the current module:

```
Jess> (deftemplate bus (slot route-number))

TRUE

Jess> (defrule take-the-bus
  ?bus <- (bus (route-number 76))
  (have-correct-change)
  =>
  (get-on ?bus))

TRUE

Jess> (ppdefrule take-the-bus)

"(defrule COMMUTE::take-the-bus
  ?bus <- (bus (route-number 76))
```

```
  (have-correct-change)
  =>
  (get-on ?bus))"
```

You can set the current module explicitly using the `set-current-module` function. The implied template `have-correct-change` was created in the COMMUTE module, because that's where the rule was defined.

6.22.2. Modules, scope, and name resolution

A module defines a *namespace* for templates and rules. This means that two different modules can each contain a rule with a given name without conflicting -- i.e., rules named `MAIN::initialize` and `COMMUTE::initialize` could be defined simultaneously and coexist in the same program. Similarly, the templates `COMPUTER::bus` and `COMMUTE::bus` could both be defined. Given this fact, there is the question of how Jess decides which template the definition of a rule or query is referring to.
When Jess is compiling a rule or deffacts definition, it will look for templates in three places, in order:

1. If a pattern explicitly names a module, only that module is searched.
2. If the pattern does not specify a module, then the module in which the rule is defined is searched first.
3. If the template is not found in the rule's module, the module MAIN is searched last. Note that this makes the MAIN module a sort of global namespace for templates.

The following example illustrates each of these possibilities:

```
Jess> (assert (MAIN::mortgage-payment 2000))

<Fact-0>

Jess> (defmodule WORK)

TRUE

Jess> (deftemplate job (slot salary))

TRUE

Jess> (defmodule HOME)

TRUE

Jess> (deftemplate hobby (slot name) (slot income))

TRUE

Jess> (defrule WORK::quit-job
  (job (salary ?s))
  (HOME::hobby (income ?i&:(> ?i (/ ?s 2))))
  (mortgage-payment ?m&:(< ?m ?i))
  =>
  (call-boss)
  (quit-job))

TRUE

Jess> (ppdefrule WORK::quit-job)
```

```
"(defrule WORK::quit-job
   (job (salary ?s))
   (HOME::hobby (income ?i&:(> ?i (/ ?s 2))))
   (MAIN::mortgage-payment ?m&:(< ?m ?i))
   =>
   (call-boss)
   (quit-job))"
```

In this example, three deftemplates are defined in three different modules: `MAIN::mortgage-payment`, `WORK::job`, and `HOME::hobby`. Jess finds the WORK::job template because the rule is defined in the WORK module. It finds the HOME::hobby template because it is explicitly qualified with the module name. And the MAIN::mortgage-payment template is found because the MAIN module is always searched as a last resort if no module name is specified.
Commands which accept the name of a construct as an argument (like `ppdefrule`, `ppdeffacts`, etc) will search for the named construct in the same way as is described above.
Note that many of the commands that list constructs (`facts`, `list-deftemplates`, `rules`, etc) accept a module name or "*" as an optional argument. If no argument is specified, these commands operate only on the current module. If a module name is given, they operate on the named module. If "*" is given, they operate on all modules.

6.22.3. Module focus and execution control

In the previous sections I described how modules provide a kind of namespace facility, allowing you to partition a rulebase into manageable chunks. Modules can also be used to control execution. In general, although any Jess rule can be activated at any time, only rules in the *focus module* will fire. Note that the *focus module* is independent from the *current module* discussed above.
Initially, the module MAIN has the focus:

```
Jess> (defmodule DRIVING)

TRUE

Jess> (defrule get-in-car
  =>
  (printout t "Ready to go!" crlf))

TRUE

Jess> (reset)

TRUE

Jess> (run)

0
```

In the example above, the rule doesn't fire because the DRIVING module doesn't have the focus. You can move the focus to another module using the `focus` function (which returns the name of the previous focus module:)

```
Jess> (focus DRIVING)

MAIN

Jess> (run)

Ready to go!
```

```
1
```

Note that you can call `focus` from the right-hand-side of a rule to change the focus while the engine is running.

Jess actually maintains a *focus stack* containing an arbitrary number of modules. The focus module is, by definition, the module on top of the stack. When there are no more activated rules in the focus module, it is "popped" from the stack, and the next module underneath becomes the focus module. You also can manipulate the focus stack with the functions `pop-focus` `list-focus-stack` `get-focus-stack` and `clear-focus-stack`

The example program `dilemma.clp` shows a good use of modules for execution control.

6.22.3.1. The auto-focus declaration

You can declare that a rule has the *auto-focus property*:

```
Jess> (defmodule PROBLEMS)

TRUE

Jess> (defrule crash
  (declare (auto-focus TRUE))
  (DRIVING::me ?location)
  (DRIVING::other-car ?location)
  =>
  (printout t "Crash!" crlf)
  (halt))

TRUE

Jess> (defrule DRIVING::travel
  ?me <- (me ?location)
  =>
  (printout t ".")
  (retract ?me)
  (assert (me (+ ?location 1))))

TRUE

Jess> (assert (me 1))

<Fact-1>

Jess> (assert (other-car 4))

<Fact-2>

Jess> (focus DRIVING)

MAIN

Jess> (run)

...Crash!
4
```

When an auto-focus rule is activated, the module it appears in is automatically pushed onto the focus stack and becomes the focus module. Modules with auto-focus rules make great "background tasks."

6.22.3.2. Returning from a rule RHS

If the function `return` is called from a rule's right-hand-side, it immediately terminates the execution of that rule's RHS. Furthermore, the current focus module is popped from the focus stack.

This suggests that you can call a module like a subroutine. You call the module from a rule's RHS using `focus` and you return from the call using `return`.

To stop executing a rule's actions without popping the focus stack, use `break` instead.

Finally, note that the auto-focus declaration can be applied to defmodules too; an auto-focus module is equivalent to a regular module in which every rule has the auto-focus property.

6.22.4. Removing modules

This should not be a common requirement, but you can remove a module from the engine using the `jess.Rete.removeDefmodule(String)` method. You might want to do this during interactive development or experimentation. You won't be able to remove a module if it's in use. If there are any templates or rules defined in the module, you'll get an exception.

7. Querying Working Memory

Jess's working memory is similar to a database; it's filled with indexed, structured data. Most of the time, you'll access working memory by pattern matching from a rule. But once in a while, you'll write some procedural code that needs to extract data from working memory directly. This chapter tells you how.

7.1. Linear search

A time-honored way to search a collection of items is by using linear search and a *filter*, a Boolean function that indicates whether an item should be part of the search result or not. Jess supports this kind of brute-force search of Java objects in working memory using the `jess.Filter` interface. You can implement `jess.Filter` and then pass an instance of your filter to the `jess.Rete.getObjects(jess.Filter)` method, which will return an `java.util.Iterator` over the selected objects. Although this might not be practical for large working memory sizes, it's certainly convenient. In the following, we run a Jess program that deals in `Payment` objects, then query working memory for all the `Payment` objects and call `process()` on each one:

```
import jess.*;
import java.util.Iterator;
public class ExMyFilter {
    interface Payment { void process(); }
    public static void main(String[] argv) throws JessException {
        Rete engine = new Rete();
        engine.batch("cashier.clp");
        Iterator it = engine.getObjects(new Filter() {
            public boolean accept(Object o) {
                return o instanceof Payment;
            }
        });
        while (it.hasNext()) {
            ((Payment) it.next()).process();
        }
    }
}
```

7.2. The defquery construct

While linear search is convenient, it's inefficient. The `defquery` construct lets you create a special kind of rule with no right-hand-side. While normal rules act spontaneously, queries are used to search the working memory under direct program control. Whereas a rule is activated once for each matching set of facts, a query gives you a `jess.QueryResult` object which gives you access to a list of all the matches.

Using a defquery involves three steps:

1. Writing the query
2. Invoking the query
3. Using the results

In the following section, we'll examine a specific example of how this is done.

7.3. A simple example

An example should make this clear. We'll go through the three steps listed above, showing one or more ways to perform each step.

7.3.1. Writing the query

Once again, we're writing a program that deals with a database of people. Once again, we'll use the following deftemplate

```
Jess> (deftemplate person (slot firstName) (slot lastName) (slot age))
```

Imagine that we want to be able to query working memory to find people with a given last name. Once we've found a person, we'll want to know their first name as well, and their age. We can write a query which lets us specify the last name we're interested in, and also lets us easily recover the first name and age of each match.

A query looks a lot like the left-hand side of a rule. We write a pattern which matches the facts that we're interested in. We *declare* the variable `?ln`, which makes it a parameter to the query, and we also include variables `?fn` and `?age` bound to the `firstName` and `age` slots, respectively.

```
Jess> (defquery search-by-name
  "Finds people with a given last name"
  (declare (variables ?ln))
  (person (lastName ?ln) (firstName ?fn) (age ?age)))
```

While we're at it, let's define some facts for the query to work with and load them into working memory:

```
Jess> (deffacts data
  (person (firstName Fred)   (lastName Smith)    (age 12))
  (person (firstName Fred)   (lastName Jones)    (age 9))
  (person (firstName Bob)    (lastName Thomas)   (age 32))
  (person (firstName Bob)    (lastName Smith)    (age 22))
  (person (firstName Pete)   (lastName Best)     (age 21))
  (person (firstName Pete)   (lastName Smith)    (age 44))
  (person (firstName George) (lastName Smithson) (age 1))
  )

Jess> (reset)
```

7.3.2. Invoking the query

After we define a query, we can call it using the `run-query*` function in Jess, or the `jess.Rete.runQueryStar(java.lang.String, jess.ValueVector)` method in Java. In both cases, we need to pass the last name we're interested in as the query parameter. In Jess, this looks like

```
Jess> (reset)

Jess> (bind ?result (run-query* search-by-name Smith))
```

The arguments to `run-query*` are the name of the query and values for each parameter. The `run-query*` function returns a `jess.QueryResult` object, which we store in the variable `?result`.

In Java, the equivalent code would look like:

```
    Rete engine = ...
    QueryResult result = engine.runQueryStar("search-by-name", new
ValueVector().add("Smith"));
```

7.3.3. Using the Results

Now that we've created a `jess.QueryResult`, it's time to iterate over all the matches and process them however we'd like. For this example, let's just print a table of the people including their full names and ages.

The interface of the `jess.QueryResult` class is very similar to that of `java.sql.ResultSet`. We use `jess.QueryResult.next()` to advance to the next result, and we use the set of `getXXX()` methods to return the values bound to the variables defined in the query. Then the following Jess code will print the output shown:

```
Jess> (while (?result next)
    (printout t (?result getString fn) " " (?result getString ln)
                ", age " (?result getInt age) crlf))

Fred Smith, age 12
Bob Smith, age 22
Pete Smith, age 44
FALSE
```

because there are three people named Smith in working memory.

The same loop in Java would look like

```
while (result.next()) {
    System.out.println(result.getString("fn") + " " + result.getString("ln")
                       + ", age" + result.getInt("age"));
}
```

7.3.4. One more time, in Java

For the sake of completeness, here's the whole program in Java. The file "query.clp" is assumed to contain the `deftemplate`, `defquery`, and `deffacts` from above.

```
import jess.*;

public class ExQuery {
    public static void main(String[] argv) throws JessException {
        Rete engine = new Rete();
        engine.batch("query.clp");
        engine.reset();
```

```
        QueryResult result =
            engine.runQueryStar("search-by-name", new ValueVector().add("Smith"));
        while (result.next()) {
            System.out.println(result.getString("fn") + " " + result.getString("ln")
                               + ", age" + result.getInt("age"));
        }
    }
}
C:\> java ExQuery
```

7.4. The variable declaration

You have already realized that two different kinds of variables can appear in a query: those that are "internal" to the query, like `?age` in the query above, and those that are "external", or to be specified in the `run-query*` command when the query is executed, like `?ln`. Jess assumes all variables in a query are internal by default; you must declare any external variables explicitly using the syntax

```
(declare (variables ?X ?Y ...))
```

which is quite similar to the syntax of a rule salience declaration.

7.5. The max-background-rules declaration

It can be convenient to use queries as triggers for backward chaining. For this to be useful, `jess.Rete.run()` must be called while the query is being evaluated, to allow the backward chaining to occur. Facts generated by rules fired during this run may appear as part of the query results. (If this makes no sense whatsoever to you, don't worry about it; just skip over this section for now.)

By default, no rules will fire while a query is being executed. If you want to allow backward chaining to occur in response to a query, you can use the `max-background-rules` declaration -- i.e.,

```
(declare (max-background-rules 10))
```

would allow a maximum of 10 rules to fire while this particular query was being executed.

7.6. The count-query-results command

To obtain just the number of matches for a query, you can use the `count-query-results` function. This function accepts the same arguments as `run-query*`, but just returns an integer, the number of matches.

7.7. Using dotted variables

The dotted variable syntax provides a convenient alternative for accessing the slots of a fact resulting from a query. Let's look at the rewrite the `search-by-name` query again. This time we'll bind a single variable to the entire fact, and use dotted variables to retrieve the results.

```
Jess> (defquery search-by-name
  "Finds people with a given last name"
  (declare (variables ?ln))
  ?person <- (person (lastName ?ln)))
```

Running the query doesn't change:

```
Jess> (reset)

Jess> (bind ?result (run-query* search-by-name Smith))
```

This time, we are going to use different code to retrieve the data:

```
Jess> (while (?result next)
    (bind ?p (?result getObject person))
    (printout t ?p.firstName " " ?p.lastName ", age " ?p.age crlf))


Fred Smith, age 12
Bob Smith, age 22
Pete Smith, age 44
FALSE
```

The `bind` stores the `jess.Fact` object that is retrieved from the current value of `?result` into variable `?p`. This, then, is conveniently accessed for its slot values `firstName`, `lastName` and `age`.

8. Using Java from Jess

8.1. Java reflection

Jess includes a number of functions that let you create and manipulate Java objects directly from Jess. Using them, you can do virtually anything you can do from Java code, including defining new classes. Here is an example in which I create a Java `HashMap` and add a few String objects to it, then lookup one object and display it. To do this, I use Jess's `new` and `call` functions:

```
Jess> (bind ?ht (new java.util.HashMap))
<Java-Object:java.util.HashMap>
Jess> (call ?ht put "key1" "element1")


Jess> (call ?ht put "key2" "element2")


Jess> (call ?ht get "key1")
"element1"
```

As you can see, Jess converts freely between Java and Jess types when it can. Java objects that can't be represented as a Jess type are called *Java object values*. The `HashMap` in the example above is one of these.
Most of the time you can omit the `call`, leading to a notation a little more like Java code -- i.e.,

```
Jess> (bind ?ht (new java.util.HashMap))
<Java-Object:java.util.HashMap>
Jess> (?ht put "key1" "element1")


Jess> (?ht put "key2" "element2")


Jess> (?ht get "key1")
"element1"
```

Jess can also access member variables of Java objects using the same `set` and `get` functions; if these functions don't find a JavaBeans property by the given name, they'll look for an instance variable next. There are also functions named `set-member` and `get-member` functions that will only work with instance variables; these can be handy if a class has instance variables and accessor methods by the same name (like the `x` member and `getX()` method in `Point`:)

```
Jess> (bind ?pt (new java.awt.Point))
<Java-Object:java.awt.Point>
```

```
Jess> (set-member ?pt x 37)

37

Jess> (set-member ?pt y 42)

42

Jess> (get-member ?pt x)

37
```

You can access static members by using the name of the class instead of an object as the first argument to these functions.

```
Jess> (get-member System out)

<Java-Object:java.io.PrintStream>
```

Note that we don't have to say "java.lang.System". The java.lang package is implicitly "imported" much as it is in Java code. Jess also has an `import` function that you can use explicitly.

Sometimes you might have trouble calling overloaded methods -- for example, passing the String "TRUE" to a Java method that is overloaded to take either a boolean or a String. In this case, you can always resort to using an explicit wrapper class -- in this case, passing a `java.lang.Boolean` object should fix the problem.

To learn more about the syntax of `call`, `new`, `set-member`, `get-member`, and other Java integration functions, see the Jess function guide.

8.1.1. Calling static methods

You can invoke static methods using `call` as well by specifying the name of the class as the first argument rather than a Java object. When you're calling a static method, "call" is not optional.

```
Jess> (call System gc)
```

There's a problem with using `call` with static methods, though: it can be ambiguous if a static method has the same name as an instance method of the `java.lang.String` class. In practice, this doesn't happen often, but when it does, it can be confusing. Therefore, in Jess 7, the preferred mechanism for invoking Java static methods is *not* to use `call`, but instead to use Jess's `import` function to import the method's class. This will automatically create a Jess function that invokes the given static method. For example:

```
Jess> (import java.util.Calendar)


Jess> (bind ?cal (Calendar.getInstance))

<Java-Object:java.util.GregorianCalendar>
```

8.1.2. Type conversion of arguments

Jess converts values from Java to Jess types according to the following table.

Java type	Jess type
A null reference	The symbol 'nil'
A void return value	The symbol 'nil'
String	RU.STRING
An array	A Jess list
boolean or java.lang.Boolean	The symbols 'TRUE' and 'FALSE'
byte, short, int, or their wrappers	RU.INTEGER
long or Long	RU.LONG
double, float or their wrappers	RU.FLOAT
char or java.lang.Character	RU.SYMBOL
anything else	RU.JAVA_OBJECT

Jess converts values from Jess to Java types with some flexibility, according to this table. Generally when converting in this direction, Jess has some idea of a *target type;* i.e., Jess has a `java.lang.Class` object and a `jess.Value` object, and wants to turn the `Value`'s contents into something assignable to the type named by the `Class`. Hence the symbol 'TRUE' could be passed to a function expecting a boolean argument, or to one expecting a String argument, and the call would succeed in both cases.

Jess type	Possible Java types
RU.JAVA_OBJECT	The wrapped object
The symbol 'nil'	A null reference
The symbols 'TRUE' or 'FALSE'	java.lang.Boolean or boolean
RU.SYMBOL, RU.STRING	String, char, java.lang.Character
RU.FLOAT	float, double, and their wrappers
RU.INTEGER	long, short, int, byte, char, and their wrappers
RU.LONG	long, short, int, byte, char, and their wrappers
RU.LIST	A Java array

8.2. Transferring values between Jess and Java code

This section describes a very easy-to-use mechanism for communicating inputs and results between Jess and Java code.
These methods are available in the class `jess.Rete`:

```
public Value store(String name, Value val);
public Value store(String name, Object val);
public Value fetch(String name);
public void clearStorage();
```

while these functions are available in Jess:

```
(store <name> <value>)
(fetch <name>)
(clear-storage)
```

Both `store` methods accept a "name" and a value (in Java, either in the form of a `jess.Value` object or an ordinary Java object; in Jess, any value), returning any old value with that name, or null (or nil in Jess) if there is none. Both `fetch` methods accept a name, and return any value stored under that name, or null/nil if there is no such object. These functions therefore let you transfer data between Jess and Java that cannot be represented textually (of course they work for Strings, too.) In this example we create an object in Java, then pass it to Jess to be used as an argument to the `list` command.

```
import jess.*;
public class ExFetch {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();
    r.store("DIMENSION", new java.awt.Dimension(10, 10));
    r.eval("(bind ?list (list dimension (fetch DIMENSION)))");
    r.eval("(printout t ?list)");
  }
}
C:\> java ExFetch

(dimension <Java-Object:java.awt.Dimension>)
```

Note that storing a null (or nil) value will result in the given name being removed from the hashtable altogether. `clearStorage()` and `clear-storage` each remove all data from the hashtable.
Note that the Jess `clear` and Java `clear()` functions will call `clearStorage()`, but `reset` and `reset()` will not. Stored data is thus available across calls to `reset()`.

8.3. Implementing Java interfaces with Jess

Many Java libraries expect you to use *callbacks*. A callback is an object that implements a specific interface; you pass it to a library method, and the methods of the callback are invoked at some expected time. GUIs work this way -- the callbacks are called *event handlers.* But Java Threads work this way too -- a `java.lang.Runnable` is a callback that gets invoked on a new thread. So being able to implement an interface is an important part of programming in Java.

Jess supports creating callbacks with the `implement` function. This function is simple to use: you tell it the name of an interface, and the name of a deffunction, and it returns an object that implements that interface by calling that function. When any method of that interface is invoked, the deffunction will be called. The first argument will be the name of the interface function, and all the arguments of the interface function will follow.

As an example, here is an implementation of `java.util.Comparator` which sorts Strings in case-insensitive order:

```
Jess> (import java.util.Comparator)

Jess> (deffunction compare(?name ?s1 ?s2)
    (return ((?s1 toUpperCase) compareTo (?s2 toUpperCase))))

TRUE

Jess> (bind ?c (implement Comparator using compare))
```

8.3.1. Lambda expressions

There's a nice shortcut you can use when calling `implement`. Instead of defining a separate `deffunction`, you can define one in-line using the `lambda` facility. The `lambda` function lets you define a deffunction without naming it, and without adding it to a Rete engine. We can redo the example above using `lambda` like this:

```
Jess> (import java.util.Comparator)

Jess> (bind ?c (implement Comparator using (lambda (?name ?s1 ?s2)
    (return ((?s1 toUpperCase) compareTo (?s2 toUpperCase))))))
```

8.4. Java Objects in working memory

You can let Jess pattern-match on Java objects using `definstance`. You can also easily put Java objects into the slots of other Jess facts, as described elsewhere in this document. This section describes some minimal requirements for objects used in either of these ways.
Jess may call the `equals` and `hashCode` methods of any objects in working memory. As such, it is very important that these methods be implemented properly. The Java API documentation lists some important properties of `equals` and `hashCode`, but I will reiterate the most important (and most often overlooked) one here: if you write `equals`, you probably *must* write `hashCode` too. For any pair of instances of a class for which `equals` returns `true`, `hashCode` must return the same value for both instances. If this rule is not observed, Jess will appear to malfunction when processing facts containing these improperly defined objects in their slots. In particular, rules that should fire may not do so.

A *value object* is an instance of a class that represents a specific value. They are often immutable like Integer, Double, and String. For Jess's purposes, a value object is one whose `hashCode()` method returns a constant -- i.e., whose hash code doesn't change during normal operation of the class. Integer, Double, and all the other wrapper classes qualify, as does String, and generally all immutable classes. Any class that doesn't override the default `hashCode()` method also qualifies as a value object by the definition. Java's Collection classes (Lists, Maps, Sets, etc.) are classic examples of classes that are *not* value objects, because their hash codes depend on the collection's contents.

As far as Jess is concerned, an object is a value object as long as its hash code won't change while the object is in working memory. This includes the case where the object is contained in a slot of any fact. If the hash code will only change during calls to `modify`, then the object is still a value object.

Jess can make certain assumptions about value objects that lead to large performance increases during pattern matching. Because many classes are actually value classes by Jess's

broad definition, Jess now assumes that all objects (except for Maps and Collections) are value objects by default. If you're working with a class that is *not* a value class, it's very important that you tell Jess about it by using the `set-value-class` function. Failure to do so will lead to undefined (bad) behavior.

8.5. Setting and Reading Java Bean Properties

As mentioned previously, Java objects can be explicitly pattern-matched on the LHS of rules, but only to the extent that they are *Java Beans*. A Java Bean is really just a Java object that has a number of methods that obey a simple naming convention for *Java Bean properties.* A class has a Bean property if, for some string *X* and type *T* it has either or both of:

- A method named `get`*X* which returns *T* and accepts no arguments; or, if *T* is boolean, named `is`*X* which accepts no arguments;
- A method named `set`X which returns void and accepts a single argument of type *T*.

Note that the capitalization is also important: for example, for a method named isVisible, the property's name is *visible*, with a lower-case v. Only the capitalization of the first letter of the name is important. You can conveniently set and get these properties using the Jess `set` and `get` methods. Note that many of the trivial changes in Java 1.1 were directed towards making most visible properties of objects into Bean properties.
An example: AWT components have many Bean properties. One is *visible*, the property of being visible on the screen. We can read this property in two ways: either by explicitly calling the `isVisible()` method, or by querying the Bean property using `get`.

```
Jess> (defglobal ?*frame* = (new java.awt.Frame "Frame Demo"))

TRUE

Jess> ;; Directly call 'isVisible', or...
(printout t (call ?*frame* isVisible) crlf)

FALSE

Jess> ;; ... equivalently, query the Bean property
(printout t (get ?*frame* visible) crlf)

FALSE
```

9. Jess Application Design

9.1. What makes a good Jess application?

Jess is a rule engine - a special kind of software that very efficiently applies rules to data. A rule-based program can have hundreds or even thousands of *rules*, and Jess will continually apply them to data stored in its *working memory*. The rules might represent the knowledge of a human expert in some domain, a set of business rules, a technical specification, or the rules of a game. The working memory might represent the state of an evolving situation (an interview, an emergency), the contents of a database, the status of a Web user's session, or a commercial account.

Many articles, and indeed, entire books have been written about rule-based systems and how to design them. It's worth reading some of this literature to give yourself an idea of what rule engines can and can't do. The most important piece of advice I can give you is that if you already know how to solve a problem procedurally, then do it that way. Don't use a rule engine and then try to force it to execute a specific sequence of rules. The whole point of a rule engine is that it's good at finding the right sequence of rules to apply, all by itself.

9.2. Command-line, GUI, or embedded?

As we've discussed, Jess can be used in many ways. Besides the different categories of problems Jess can be applied to, being a library, it is amenable to being used in many different kinds of Java programs. Jess can be used in command-line applications, GUI applications, servlets, and applets. Furthermore, Jess can either provide the Java `main()` for your program, or you can write it yourself. You can develop Jess applications (with or without GUIs) without compiling a single line of Java code. You can also write Jess applications which are controlled entirely by Java code you write, with a minumum of Jess language code.

The most important step in developing a Jess application is to choose an architecture from among the almost limitless range of possibilities. One way to organize the possibilities is to list them in increasing order of the amount of Java programming involved.

1. Pure Jess language scripts. No Java code at all.
2. Pure Jess language scripts, but the scripts access Java APIs.
3. Mostly Jess language scripts, but some custom Java code in the form of new Jess commands written in Java.
4. Half Jess language scripts, with a substantial amount of Java code providing custom commands and APIs; `main()` provided by Jess.
5. Half Jess language scripts, with a substantial amount of Java code providing custom commands and APIs; `main()` provided by you or an application server.
6. Mostly Java code, which loads Jess language scripts at runtime.
7. All Java code, which manipulates Jess entirely through its Java API.

Examples of some of these types of applications are packaged with Jess. The basic examples like `wordgame.clp`, `zebra.clp`, and `fullmab.clp` are all type 1) programs. `awtdraw.clp`, `swingdraw.clp` and `jframe.clp` are type 2) programs. The example in this chapter is a type 6 application.

Your choice can be guided by many factors, but ultimately it will depend on what you feel most comfortable with. Types 4) and 5) are most prevalent in real-world applications.

10. Introduction to Programming with Jess in Java

There are two main ways in which Java code can be used with Jess: Java can be used to extend Jess, and the Jess library can be used from Java. The material in this section is relevant to both of these endeavors. Refer to the API documentation for the complete story on these classes.
Note: the code samples herein are necessarily not complete Java programs. In general, all excerpted code would need to appear inside a try block, inside a Java method, inside a Java class, to compile; and all Java source files are expected to include the `"import jess.*;"` declaration. Sometimes examples build on previous ones; this is usually clear from context. Such compound examples will need to be assembled into one method before compiling.

10.1. The jess.Rete class

The `jess.Rete` class is the rule engine itself. Each `jess.Rete` object has its own working memory, agenda, rules, etc. To embed Jess in a Java application, you'll simply need to create one or more `jess.Rete` objects and manipulate them appropriately. We'll cover this in more detail in the section on embedding Jess in Java applications. Here I will cover some general features of the `jess.Rete` class.

10.1.1. Equivalents for common Jess functions

Several of the most commonly used Jess functions are wrappers for methods in the `jess.Rete` class. Examples are `run()`, `run(int)`, `reset()`, `clear()`, `assertFact(Fact)`, `retract(Fact)`, `retract(int)`, and `halt()`. You can call these from Java just as you would from Jess.

10.1.2. Executing other Jess commands

You can use the `Rete` class's `jess.Rete.eval(java.lang.String)` method to easily execute, from Java, any Jess function call or construct definition that can be represented as a parseable String. For example,

```
import jess.*;
public class ExSquare {
  public static void main(String[] unused) {
    try {
        Rete r = new Rete();
        r.eval("(deffunction square (?n) (return (* ?n ?n)))");
        Value v = r.eval("(square 3)");

        // Prints '9'
        System.out.println(v.intValue(r.getGlobalContext()));
    } catch (JessException ex) {
        System.err.println(ex);
    }
  }
}
C:\> java ExSquare

9
```

`eval()` returns the `jess.Value` object returned by the command. Commands executed via `eval()` may refer to Jess variables; they will be interpreted in the *global context*. In general, only defglobals can be used in this way.
Note that you may only pass one function call or construct at a time to `eval()`.

10.1.2.1. Optional commands

Note that when you create a Rete object from Java, it will already contain definitions for all of the functions that come with Jess. There are no longer any "optional" commands.

10.1.3. The script library

Some of Jess's commands are defined in Jess language code, in the file `jess/scriptlib.clp`. Each `Rete` object will load this script library when it is created and again if (clear) is called. In previous versions of Jess you had to do this yourself; this is no longer necessary.

10.1.4. Methods for adding, finding and listing constructs

The easiest (and still encouraged) way to define templates, `defglobals`, and other constructs is to use Jess language code and let Jess parse the textual definition. However, many of these constructs are represented by public classes in the Jess library, and if you wish, you can construct your own instances of these in Java code and add them to an engine explicitly. This is currently possible for most, but not all, Jess constructs. Right now the `jess.Defrule` class does not expose enough public methods to properly create one outside of the `jess` package. This is deliberate, as this API is likely to change again in the near future. For information about the classes mentioned here (`jess.Deftemplate`, `jess.Defglobal`, etc) see the API documentation.

These `jess.Rete` methods let you add constructs to the engine:

- `public void addDeffacts(Deffacts)`
- `public void addDefglobal(Defglobal)`
- `public void addDefmodule(Defmodule)`
- `public void addDefrule(Defrule)`
- `public void addDeftemplate(Deftemplate)`
- `public void addUserfunction(Userfunction)`
- `public void addUserpackage(Userpackage)`

These methods return individual constructs from within the engine, generally by name:

- `public Defglobal findDeffacts(String)`
- `public Defglobal findDefglobal(String)`
- `public Defrule findDefrule(String)`
- `public Deftemplate findDeftemplate(String)`
- `public Userfunction findUserfunction(String)`

These methods return `java.util.Iterator`s of various data structures in the engine:

- `public Iterator listActivations()`

- `public Iterator listDeffacts()`
- `public Iterator listDeffunctions()`
- `public Iterator listDefglobals()`
- `public Iterator listDefrules()`
- `public Iterator listDeftemplates()`
- `public Iterator listFacts()`
- `public Iterator listFunctions()`
- `public Iterator listModules()`

Note that the utility class `jess.FilteringIterator` is very convenient for use together with these methods. Together with a `jess.Filter` implementation, this class lets you create an Iterator that silently discards objects that don't match some criteria. So, for example, you could use the built-in filter `jess.Filter$ByModule` to iterate over only the facts in a given module:

```
import jess.*;
import java.util.*;

public class ExFilter {
    public static void main(String[] argv) throws JessException {
        // Run a Jess program...
        Rete engine = new Rete();
        engine.batch("somecode.clp");

        // ... now retrieve only the facts in a module named RESULTS
        Iterator it = new FilteringIterator(engine.listFacts(),
                                            new Filter.ByModule("RESULTS"));
    }
}
```

10.1.5. I/O Routers

Several Jess functions like `printout`, `format`, `read`, and `readline` take an *I/O router name* as an argument, while other functions like `open` return an I/O router name. An I/O router name is just a symbolic name for a Java `java.io.Writer` and/or a `java.io.Reader`. Each `jess.Rete` instance keeps separate tables of input routers and output routers, so that both a Reader and a Writer can be registered under the same name in each rule engine. When you call, for example, `printout`, Jess uses the first argument to look up the appropriate Writer in that table, and that's where the output will go.

The most commonly used router is `t`, which is used as Jess' standard input and output. Jess also has a built-in router named `WSTDOUT` for printing user messages internally -- for example, the Jess> prompt and the output of commands like `facts` and `ppdefrule`. The `read` and `readline` commands take input from the `t` router by default. Output from the `watch` function goes to the `WSTDOUT` router by default, but you can make it go to any other router using the `jess.Rete.setWatchRouter(java.lang.String)` method.

As startup, Jess's standard routers are connected to Java's standard streams, so that output goes to the command-line window. This is perfect for command-line programs, but of course not acceptable for GUI-based applications. To remedy this, Jess lets you connect the `t` router (or any other router) to any Java `java.io.Reader` and `java.io.Writer` objects you choose. In fact,

you can not only redirect the built-in routers, but you can add routers of your own, in much the same way that the `open` command creates a new router that reads from a file.

These functions in the Rete class let you manipulate the router list:

- `public void addInputRouter(String s, Reader is, boolean consoleLike)`
- `public void addOutputRouter(String s, Writer os)`
- `public Reader getInputMode(String s)`
- `public Reader getInputRouter(String s)`
- `public Writer getOutputRouter(String s)`
- `public void removeInputRouter(String s)`
- `public void removeOutputRouter(String s)`
- `public void setWatchRouter(String s)`

The words "input" and "output" are from the perspective of the Jess library itself; i.e., Jess reads from input routers and writes to output routers.

Note that you can use the same name for an input router and an output router (the `t` router is like that.) Note also that although these functions accept and return generic `Reader` and `Writer` objects, Jess internally uses `java.io.PrintWriter` and `java.io.BufferedReader`. If you pass in other types, Jess will construct one of these preferred classes to "wrap" the object you pass in.

When Jess starts up, there are three output routers and one input router defined: the `t` router, which reads and writes from the standard input and output; the `WSTDOUT` router, which Jess uses for all prompts, diagnostic outputs, and other displays; and the `WSTDERR` router, which Jess uses to print stack traces and error messages. By default, `t` is connected to `System.in` and `System.out`, and both `WSTDOUT` and `WSTDERR` are connected to `System.out` (neither is connected to `System.err`.) You can reroute these inputs and outputs simply by changing the Readers and Writers they are attached to using the above functions. You can use any kind of streams you can dream up: network streams, file streams, etc.

The `boolean` argument `consoleLike` to the `addInputRouter` method specifies whether the stream should be treated like the standard input or like a file. The difference is that on console-like streams, a `read` call consumes an entire line of input, but only the first token is returned; while on file-like streams, only the characters that make up each token are consumed on any one call. That means, for instance, that a `read` followed by a `readline` will consume two lines of text from a console-like stream, but only one from a file-like stream, given that the first line is of non-zero length.

The `jess.Rete` class has two more handy router-related methods: `getOutStream()` and `getErrStream()`, both of which return a `java.io.PrintWriter` object. `getOutStream()` returns a stream that goes to the same place as the current setting of `WSTDOUT`; `getErrStream()` does the same for `WSTDERR`.

10.1.6. TextAreaWriter, JTextAreaWriter and TextReader

Jess ships with three utility classes that can be very useful when building GUIs for Jess: the `jess.awt.TextAreaWriter`, `jess.swing.JTextAreaWriter` and `jess.awt.TextReader` classes. All three can serve as adapters between Jess and graphical input/output widgets. The

TextAreaWriter class is, as the name implies, a Java java.io.Writer that sends any data written to it to a java.awt.TextArea. This lets you place Jess's output in a scrolling window on your GUI. The jess.Console and jess.ConsoleApplet Jess GUIs use these classes. To use TextAreaWriter simply call addOutputRouter(), passing in an instance of this class:

```
import java.awt.TextArea;
import jess.awt.*;
import jess.*;
public class ExTAW {
  public static void main(String[] unused) throws JessException {
     TextArea ta = new TextArea(20, 80);
     TextAreaWriter taw = new TextAreaWriter(ta);

     Rete r = new Rete();
     r.addOutputRouter("t", taw);
     r.addOutputRouter("WSTDOUT", taw);
     r.addOutputRouter("WSTDERR", taw);
     // Do something interesting, then...
     System.exit(0);
  }
}
C:\> java ExTAW
```

Now the output of the printout command, for example, will go into a scrolling window (of course, you need to display the TextArea on the screen somehow!) Study jess/ConsolePanel.java and jess/Console.java to see a complete example of this. JTextAreaWriter works the same way, but using a Swing javax.swing.JTextArea instead. jess.awt.TextReader is similar, but it is a java.io.Reader instead. It is actually quite similar to java.io.StringReader, except that you can continually add new text to the end of the stream (using the appendText() method). It is intended that you create a jess.awt.TextReader, install it as an input router, and then (in an AWT event handler, somewhere) append new input to the stream whenever it becomes available. See the same jess/Console* files for a complete usage example for this class as well.

10.2. The jess.JessException class

The jess.JessException exception type is the only kind of exception thrown by any functions in the Jess library. jess.JessException is rather complex, as exception classes go. An instance of this class can contain a wealth of information about an error that occurred in Jess. Besides the typical error message, a jess.JessException may be able to tell you the name of the routine in which the error occurred, the name of the Jess constructs that were on the exceution stack, the relevant text and line number of the executing Jess language program, and the Java exception that triggered the error (if any.) See the the API documentation for details.

One of the most important pieces of advice for working with the Jess library is that in your catch clauses for JessException, *display the exception object*. Print it to System.out, or convert to a String and display it in a dialog box. The exceptions are there to help you by telling when something goes wrong; don't ignore them.

Another important tip: the JessException class has a method getCause which returns non-null when a particular JessException is a wrapper for another kind of exception. For example, if you use the Jess function call to call a function that throws an exception, then call will throw a JessException, and calling JessException.getCause() will return the real exception that was thrown. Your JessException handlers should always check getCause(); if your handler simply displays a thrown exception, then it should display the return value of getCause(), too. getCause() replaces the now deprecated getNextException().

10.3. The jess.Value class

The class `jess.Value` is probably the one you'll use the most in working with Jess. A `Value` is a self-describing data object. Every datum in Jess is contained in one. Once it is constructed, a `Value`'s type and contents cannot be changed; it is *immutable.* `Value` supports a `type()` function, which returns one of these type constants (defined in the class `jess.RU` (RU = "Rete Utilities")):

```
final public static int NONE            =      0; // an empty value (not NIL)
final public static int SYMBOL          =      1; // a symbol
final public static int STRING          =      2; // a string
final public static int INTEGER         =      4; // an integer
final public static int VARIABLE        =      8; // a variable
final public static int FACT            =     16; // a jess.Fact object
final public static int FLOAT           =     32; // a double float
final public static int FUNCALL         =     64; // a function call
final public static int LIST            =    512; // a list
final public static int DESCRIPTOR      =   1024; // (internal use)
final public static int JAVA_OBJECT     =   2048; // a Java object
final public static int INTARRAY        =   4096; // (internal use)
final public static int MULTIVARIABLE   =   8192; // a multifield
final public static int SLOT            =  16384; // (internal use)
final public static int MULTISLOT       =  32768; // (internal use)
final public static int LONG            =  65536; // a Java long
final public static int LAMBDA          = 131072; // a lambda expression
```

Please always use the names, not the literal values, as the latter are subject to change without notice.

`Value` objects can be constructed by specifying the data and (usually) the type. Each overloaded constructor assures that the given data and the given type are compatible. Note that for each constructor, more than one value of the `type` parameter may be acceptable. The available constructors are:

```
public Value(Object o)
public Value(String s, int type) throws JessException
public Value(Value v)
public Value(ValueVector f, int type) throws JessException
public Value(double d, int type) throws JessException
public Value(int value, int type) throws JessException
public Value(boolean b)
```

10.3.1. ValueFactory

Alternatively, you can use the `jess.ValueFactory` class to obtain Value objects. This is a good idea because ValueFactory caches many of the values you obtain through it, leading to a sometimes drastic reduction in memory use due to sharing. You can get a ValueFactory object using the `jess.Rete.getValueFactory()` method:

```
import jess.*;
public class ExValueFactory {
  public static void main(String[] unused) throws JessException {
    Rete engine = new Rete();
    ValueFactory f = engine.getValueFactory();
    Value v1 = f.get("foo", RU.SYMBOL);
    Value v2 = f.get("foo", RU.SYMBOL);
```

```
    // Prints "true"
    System.out.println(v1 == v2);
  }
}
C:\> java ExValueFactory

true
```

`Value` supports a number of functions to get the actual data out of a `Value`object. These are

```
public Object javaObjectValue(Context c) throws JessException
public String stringValue(Context c) throws JessException
public String symbolValue(Context c) throws JessException
public Fact factValue(Context c) throws JessException
public Funcall funcallValue(Context c) throws JessException
public ValueVector listValue(Context c) throws JessException
public double floatValue(Context c) throws JessException
public double numericValue(Context c) throws JessException
public int intValue(Context c) throws JessException
public long longValue(Context c) throws JessException
```

The class `jess.Context` is described in the next section. If you try to convert random values by creating a Value and retrieving it as some other type, you'll generally get a JessException. However, some types can be freely interconverted: for example, integers and floats.

10.3.2. The subclasses of jess.Value

`jess.Value` has a number of subclasses: `jess.Variable`, `jess.FuncallValue`, `jess.FactIDValue`, and `jess.LongValue` are the four of most interest to the reader. When you wish to create a value to represent a variable, a function call, a fact, or a Java long, you must use the appropriate subclass.
Note to the design-minded: we should have used a Factory pattern here and hidden the subclasses from the programmer. This will be introduced in a future version of Jess.

10.3.2.1. The class jess.Variable

Use this subclass of Value when you want to create a Value that represents a Variable. The one constructor looks like this:

```
public Variable(String s, int type) throws JessException
```

The type must be `RU.VARIABLE` or `RU.MULTIVARIABLE` or an exception will be thrown. The String argument is the name of the variable, without any leading '?' or '$' characters.

10.3.2.2. The class jess.FuncallValue

Use this subclass of Value when you want to create a Value that represents a function call (for example, when you are creating a `jess.Funcall` containing nested function calls.) The one constructor looks like this:

```
public FuncallValue(Funcall f) throws JessException
```

10.3.2.3. The class jess.LongValue

Use this subclass of Value when you want to create a Value that represents a Java long. These are mostly used to pass to Java functions called via reflection. The one constructor looks like

```
public LongValue(long l) throws JessException
```

10.3.2.4. The class jess.FactIDValue

Use this subclass of Value when you want to create a Value that represents a fact-id. The one constructor looks like this:

```
public FactIDValue(Fact f) throws JessException
```

In previous versions of Jess, fact-id's were more like integers; now they are really references to facts. As such, a fact-id must represent a valid jess.Fact object. Call `javaObjectValue(Context)` to get the jess.Fact object, and call `Fact.getFactId()` to get the fact-id as an integer. This latter manipulation will now rarely, if ever, be necessary.

10.3.3. Value resolution

Some `jess.Value` objects may need to be *resolved* before use. To resolve a `jess.Value` means to interpret it in a particular context. `jess.Value` objects can represent both static values (symbols, numbers, strings) and dynamic ones (variables, function calls). It is the dynamic ones that obviously have to be interpreted in context.
All the `jess.Value` member functions, like `intValue()`, that accept a `jess.Context` as an argument are *self-resolving;* that is, if a `jess.Value` object represents a function call, the call will be executed in the given `jess.Context`, and the `intValue()` method will be called on the result. Therefore, you often don't need to worry about resolution as it is done automatically. There are several cases where you will, however.

- *When interpreting arguments to a function written in Java.* The parameters passed to a Java Userfunction may themselves represent function calls. It may be important, therefore, that these values be resolved only once, as these functions may have side-effects (I'm tempted to use the computer-science word: these functions may not be *idempotent.* Idempotent functions have no side-effects and thus may be called multiple times without harm.) You can accomplish this by calling one of the `(x)Value()` methods and storing the return value, using this return value instead of the parameter itself. Alternatively, you may call `resolveValue()`and store the return value in a new `jess.Value` variable, using this value as the new parameter. Note that the `type()` method will return `RU.VARIABLE` for a `jess.Value` object that refers to a variable, regardless of the type of the value the variable is bound to. The resolved value will return the proper type.

  Note that arguments to `deffunctions` are resolved automatically, before your Jess language code runs.

- *When returning a `jess.Value` object from a function written in Java.* If you return one of a function's parameters from a Java Userfunction, be sure to return the return value of `resolveValue()`, not the parameter itself.
- *When storing a `jess.Value` object.* It is important that any values passed out of a particular execution context be resolved; for example, before storing a `Value` object in a `Map`, `resolveValue()` should always be called on both the key and object.

10.4. The jess.Context class

`jess.Context` represents an execution context for the evaluation of function calls and the resolution of variables. There are very few public member functions in this class, and only a few of general importance.
You can use `getVariable()` and `setVariable()`to get and change the value of a variable from Java code, respectively.
The function `getEngine()` gives any `Userfunction` access to the Rete object in which it is executing.
When a `Userfunction` is called, a `jess.Context` argument is passed in as the final argument. You should pass this `jess.Context` to any `jess.Value.(x)Value()` calls that you make.

10.5. The jess.ValueVector class

The `jess.ValueVector` class is Jess's internal representation of a *list*, and therefore has a central role in programming with Jess in Java. The `jess.ValueVector` class itself is used to represent generic lists, while specialized subclasses are used as function calls (`jess.Funcall`), facts (`jess.Fact`), and templates (`Deftemplate`).

Working with `ValueVector` itself is simple. Its API is reminiscent of `java.util.Vector`. Like that class, it is a self-extending array: when new elements are added the `ValueVector` grows in size to accomodate them. Here is a bit of example Java code in which we create the Jess list `(a b c)`. Note that the `add()` method has several overloaded forms that convert primitives into `jess.Value` objects. The overload used here automatically converts its argument into a Jess symbol.

```
import jess.*;
public class ExABC {
  public static void main(String[] unused) throws JessException {
    ValueVector vv = new ValueVector();
    vv.add("a");
    vv.add("b");
    vv.add("c");

    // Prints "(a b c)"
    System.out.println(vv.toStringWithParens());
  }
}
C:\> java ExABC

(a b c)
```

The `add()` function returns the `ValueVector` object itself, so that `add()` calls can be chained together for convenience:

```
import jess.*;
public class ExChain {
  public static void main(String[] unused) throws JessException {
```

```
    ValueVector vv = new ValueVector();
    vv.add("a").add("b").add("c");
    // Prints "(a b c)"
    System.out.println(vv.toStringWithParens());
  }
}
C:\> java ExChain

(a b c)
```

To pass a list from Java to Jess, you should enclose it in a `jess.Value` object of type `RU.LIST`.

10.6. The jess.Funcall class

`jess.Funcall` is a specialized subclass of `ValueVector` that represents a Jess function call. It contains the name of the function, an internal pointer to the actual `jess.Userfunction` object containing the function code, and the arguments to pass to the function.
You can call Jess functions using `jess.Funcall` if you prefer, rather than using `jess.Rete.executeFunction()`. This method has less overhead since there is no parsing to be done. This example calls Jess's "set-reset-globals" function:

```
import jess.*;
public class ExResetGlobals {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();
    Context c = r.getGlobalContext();
    Funcall f = new Funcall("set-reset-globals", r);
    f.arg(Funcall.FALSE);
    Value result = f.execute(c);
    System.out.println(result);
  }
}
C:\> java ExResetGlobals

FALSE
```

The example shows several styles of using `jess.Funcall`. You can chain `add()` calls, but remember that `add()` returns `ValueVector`, so you can't call `execute()` on the return value of `Funcall.add()` A special method `arg()` is provided for this purpose; it does the same thing as `add()` but returns the `Funcall` as a `Funcall`.
The first entry in a `Funcall`'s `ValueVector` is the name of the function, even though you don't explicitly set it. Changing the first entry will not automatically change the function the `Funcall` will call!
The `Funcall` class also contains some public static constant `Value` member objects that represent the special symbols `nil`, `TRUE`, `FALSE`, `EOF`, etc. You are encouraged to use these.

10.7. The jess.Fact class

Another interesting subclass of `ValueVector` is `jess.Fact`, which, predictably, is how Jess represents facts. A `Fact` is stored as a list in which all the entries correspond to slots. The head or name of the fact is stored in a separate variable (available via the `getName()` method.)

Once you assert a jess.Fact object, you no longer "own" it - it becomes part of the Rete object's internal data structures. As such, you must not change the values of any of the Fact's slots. If you retract the fact, the Fact object is released and you are free to alter it as you wish. Alternatively, you can use the `jess.Rete.modify(jess.Fact, java.lang.String, jess.Value)` method to modify a fact.

10.7.1. Constructing an Unordered Fact from Java

In the following example, we create a template and assert an unordered fact that uses it.

```
import jess.*;
public class ExPoint {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();
    r.eval("(deftemplate point \"A 2D point\" (slot x) (slot y))");

    Fact f = new Fact("point", r);
    f.setSlotValue("x", new Value(37, RU.INTEGER));
    f.setSlotValue("y", new Value(49, RU.INTEGER));
    r.assertFact(f);

    r.eval("(facts)");
  }
}
C:\> java ExPoint

f-0   (MAIN::point (x 37) (y 49))
For a total of 1 facts in module MAIN.
```

10.7.2. Constructing a Multislot from Java

In this example, the template has a multislot. In Java, a multislot is represented by a `Value` of type `RU.LIST`; the `Value` object contains a `ValueVector` containing the fields of the multislot.

```
import jess.*;
public class ExMulti {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();
    r.eval("(deftemplate vector \"A named vector\" (slot name) (multislot list))");

    Fact f = new Fact("vector", r);
    f.setSlotValue("name", new Value("Groceries", RU.SYMBOL));
    ValueVector vv = new ValueVector();
    vv.add(new Value("String Beans", RU.STRING));
    vv.add(new Value("Milk", RU.STRING));
    vv.add(new Value("Bread", RU.STRING));
    f.setSlotValue("list", new Value(vv, RU.LIST));
    r.assertFact(f);

    r.eval("(facts)");
  }
}
C:\> java ExMulti

f-0   (MAIN::vector (name Groceries) (list "String Beans" "Milk" "Bread"))
```

```
For a total of 1 facts in module MAIN.
```

10.7.3. Constructing an Ordered Fact from Java

An ordered fact is actually represented as an unordered fact with a single slot: a multislot named `__data`. You don't need to create a template for an ordered fact: one will be created automatically if it doesn't already exist.

```
import jess.*;
public class ExOrdered {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();

    Fact f = new Fact("letters", r);
    ValueVector vv = new ValueVector();
    vv.add("a").add("b").add("c");
    f.setSlotValue("__data", new Value(vv, RU.LIST));
    r.assertFact(f);

    r.eval("(facts)");
  }
}
C:\> java ExOrdered

f-0   (MAIN::letters a b c)
For a total of 1 facts in module MAIN.
```

10.8. The jess.Deftemplate class

Yet another interesting subclass of `ValueVector` is `jess.Deftemplate`, the purpose of which should be obvious. `Deftemplate` has a fairly large interface which allows you to set and query the properties of a template's slots.
This example is an alternative to the `deftemplate` command in the previous example.

```
import jess.*;
public class ExBuildDeftemplate {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();
    Deftemplate dt = new Deftemplate("point", "A 2D point", r);
    Value zero = new Value(0, RU.INTEGER);
    dt.addSlot("x", zero, "NUMBER");
    dt.addSlot("y", zero, "NUMBER");
    r.addDeftemplate(dt);

    // Now create and assert Fact
  }
}
```

10.9. Parsing Jess code with jess.Jesp

You can parse Jess language code directly with the class `jess.Jesp`. Simply loading the contents of a file (or any other data source that can be supplied as a `java.io.Reader` is very easy:

```
import jess.*;
import java.io.*;
public class ExReadInFile {
  public static void main(String[] unused) throws JessException, IOException {
    Rete engine = new Rete();
    FileReader file = new FileReader("myfile.clp");
    try {
      Jesp parser = new Jesp(file, engine);
      parser.parse(false);
    } finally {
      file.close();
    }
  }
}
```

But `jess.Jesp`'s public interface is much richer than that. If you want to, you can parse the file one expression at a time, and have access to the parsed data. The method `parseExpression` returns `java.lang.Object`, and the returned object can either be a `jess.Value` or one of the Jess classes that represent a construct (`jess.Defrule`, `jess.Deftemplate`, etc.) In addition, you can choose to have the parser execute function calls as it parses them, or simply return them to you unexecuted (this is controlled by the second argument to `parseExpression`.

```
import jess.*;
import java.io.*;
public class ExParseExpressions {
  public static void main(String[] unused) throws JessException, IOException {
    Rete engine = new Rete();
    FileReader file = new FileReader("myfile.clp");
    Context context = engine.getGlobalContext();
    try {
      Jesp parser = new Jesp(file, engine);
      Object result = Funcall.TRUE;
      while (!result.equals(Funcall.EOF)) {
        result = parser.parseExpression(context, false);
        // Here you can use instanceof to determine what sort
        // of object "result" is, and process it however you want
      }
    } finally {
      file.close();
    }
  }
}
```

There are also methods in `jess.Jesp` to control whether comments should be returned from `parseExpression` or just skipped, methods to fetch various pieces of information about the parsing process, and a mechanism for controlling whether warnings or just errors should be reported.

10.10. The jess.Token class

The `jess.Token` class is used to represent partial matches in the Rete network. You'll use it if you're writing an Accelerator (not documented here) or perhaps if you're working with queries. Only a few methods of `jess.Token` are public, and fewer are of use to the programmer. `int size()` tells you how many `jess.Fact`s are in a given `jess.Token`. The most important method is `Fact fact(int)`, which returns the `jess.Fact` objects that make up the partial match. Its argument is the zero-based index of the `jess.Fact` to retrieve, and must be between 0 and the return value of `size()`. Each `Fact` will correspond to one pattern on a rule or query LHS; dummy facts are inserted for `not` and `test` CEs.

10.11. The jess.JessEvent and jess.JessListener classes

`jess.JessEvent` and `jess.JessListener` make up Jess's rendition of the standard Java event pattern. By implementing the `JessListener` interface, a class can register itself with a source of `JessEvent`s, like the `jess.Rete` class. `jess.Rete` (potentially) fires events at all critical junctures during its execution: when rules fire, when a `reset()` or `clear()` call is made, when a fact is asserted or retracted, etc. `JessEvent` has a `getType()` method to tell you what sort of event you have been notified of; the type will be one of the constants in the `JessEvent` class.
You can control which events a `jess.Rete` object will fire using the `setEventMask()` method. The argument is the result of logical-OR-ing together some of the constants in the `jess.JessEvent` class. By default, the event mask is 0 and no events are sent.

As an example, let's suppose you'd like your program's graphical interface to display a running count of the number of facts on the fact-list, and the name of the last executed rule. You've provided a static method, `MyGUI.displayCurrentRule(String ruleName),` which you would like to have called when a rule fires. You've got a pair of methods `MyGUI.incrementFactCount()` and `MyGUI.decrementFactCount()` to keep track of facts. And you've got one more static method, `MyGUI.clearDisplay()`, to call when Jess is cleared or reset. To accomplish this, you simply need to write an event handler, install it, and set the event mask properly. Your event handler class might look like this.

```
import jess.*;

public class ExMyEventHandler implements JessListener {
  public void eventHappened(JessEvent je) {
    int defaultMask = JessEvent.DEFRULE_FIRED | JessEvent.FACT |
                      JessEvent.RESET | JessEvent.CLEAR;
    int type = je.getType();
    switch (type) {
      case JessEvent.CLEAR:
      case JessEvent.RESET:
        // MyGUI.clearDisplay();
        break;

      case JessEvent.DEFRULE_FIRED:
        // MyGUI.displayCurrentRule( ((Activation)
je.getObject()).getRule().getName());
        break;

      case JessEvent.FACT | JessEvent.REMOVED:
        // MyGUI.decrementFactCount();
```

```
        break;

      case JessEvent.FACT:
        // MyGUI.incrementFactCount();
        break;

      default:
        // ignore
    }
  }
}
```

Note how the event type constant for fact retracting is composed from `FACT | REMOVED`. In general, constants like `DEFRULE, DEFTEMPLATE,` etc, refer to the addition of a new construct, while composing these with `REMOVE` signifies the removal of the same construct.
The `getObject()` method returns ancillary data about the event. In general, it is an instance of the type of object the event refers to; for `DEFRULE_FIRED` it is a `jess.Activation`.
To install this listener, you would simply create an instance and call `jess.Rete.addEventListener()`, then set the event mask:

```
import jess.*;
public class ExMask {
  public static void main(String[] unused) throws JessException {
    Rete engine = new Rete();
    engine.addJessListener(new ExMyEventHandler());
    engine.setEventMask(engine.getEventMask() |
                        JessEvent.DEFRULE_FIRED | JessEvent.CLEAR |
                        JessEvent.FACT | JessEvent.RESET );
  }
}
```

Note that each event handler added will have a negative impact on Jess performance so their use should be limited.
There is no way to receive only one of an event / (event | `REMOVE`) pair.

10.11.1. Working with events from the Jess language

It's possible to work with the event classes from Jess language code as well. To write an event listener, you can use the `jess.JessEventAdapter` class. This class works rather like the `jess.awt` adapter classes do. Usage is best illustrated with an example. Let's say you want to print a message each time a new template is defined, and you want to do it from Jess code. Here it is:

```
Jess> ;; make code briefer
(import jess.*)

TRUE

Jess> ;; Here is the event-handling deffunction
;; It accepts one argument, a JessEvent
(deffunction display-deftemplate-from-event (?evt)
  (if (eq (JessEvent.DEFTEMPLATE) (get ?evt type)) then
      (printout t "New deftemplate: " (call (call ?evt getObject) getName) crlf)))

TRUE

Jess> ;; Here we install the above function using a JessEventAdapter
(call (engine) addJessListener
```

```
(new JessEventAdapter display-deftemplate-from-event (engine)))

Jess> ;; Now we add DEFTEMPLATE to the event mask
(set (engine) eventMask
(bit-or (get (engine) eventMask) (JessEvent.DEFTEMPLATE)))
```

Now whenever a new template is defined, a message will be displayed.

10.12. Setting and Reading Java Bean Properties

As mentioned previously, Java objects can be explicitly pattern-matched on the LHS of rules, but only to the extent that they are *Java Beans*. A Java Bean is really just a Java object that has a number of methods that obey a simple naming convention for *Java Bean properties.* A class has a Bean property if, for some string *X* and type *T* it has either or both of:

- A method named `get`*X* which returns *T* and accepts no arguments; or, if *T* is boolean, named `is`*X* which accepts no arguments;
- A method named `set`X which returns void and accepts a single argument of type *T*.

Note that the capitalization is also important: for example, for a method named isVisible, the property's name is *visible*, with a lower-case `v`. Only the capitalization of the first letter of the name is important. You can conveniently set and get these properties using the Jess `set` and `get` methods. Note that many of the trivial changes in the Java 1.1 were directed towards making most visible properties of objects into Bean properties.
An example: AWT components have many Bean properties. One is *visible*, the property of being visible on the screen. We can read this property in two ways: either by explicitly calling the `isVisible()` method, or by querying the Bean property using `get`.

```
Jess> (defglobal ?*frame* = (new java.awt.Frame "Frame Demo"))

TRUE

Jess> ;; Directly call 'isVisible', or...
(printout t (call ?*frame* isVisible) crlf)

FALSE

Jess> ;; ... equivalently, query the Bean property
(printout t (get ?*frame* visible) crlf)

FALSE
```

10.13. Formatting Jess Constructs

The class `jess.PrettyPrinter` can produce a formatted rendering of many Jess objects, including `jess.Defrule`s, `Deffunction`s `jess.Defquery`s, etc -- anything that implements the `jess.Visitable` interface.
`jess.PrettyPrinter` is very simple to use: you just create an instance, passing the object to be rendered as a constructor argument, and then call `toString` to get the formatted result.

```
import jess.*;
public class ExPretty {
  public static void main(String[] unused) throws JessException {
    Rete r = new Rete();
    r.eval("(defrule myrule (A) => (printout t \"A\" crlf))");
    Defrule dr = (Defrule) r.findDefrule("myrule");
    System.out.println(new PrettyPrinter(dr));
  }
}
C:\> java ExPretty


(defrule MAIN::myrule
  (A)
  =>
  (printout t "A" crlf))
```

11. Embedding Jess in a Java Application

11.1. Introduction

In the first part of this chapter, we'll look at one way to embed Jess into a Java application. It's a *pricing engine* which determines the prices individual customers will pay for goods ordered through our e-commerce site. The material presented here will be easier to understand if you are familiar with Jess's Java APIs and how to write rules in the Jess programming language.

At the end of this chapter, I'll talk about some general considerations that are useful when planning an application that embeds the Jess rule engine.

11.2. Motivation

Imagine you're working on a pricing engine for online sales. The engine is supposed to look at each order, together with a customer's purchasing history, and apply various discounts and offers to the order. Imagine further that you've coded this up in a traditional Java class.

Your boss comes in, says, "Give a 10% discount to everybody who spends more than $100." Great. You add a single if-then statement, recompile, and you're back in business.

Boss comes in, says, "Give a 25% discount on items the customer buys three or more of." Another if-then, another recompile.

Boss comes in, says "Revoke 10% discount per-order, make it 10% if you spend $250 or more in one month. Also, if somebody buys a CD writer, send them a free sample of CD-RW disks; but only if they're a repeat customer. Oh, and by the way, we need to price shipping based on geographic zone, except for people with multiple locations..."

After a few weeks of this, if you've been using traditional programming techniques, your code is a gnarled mess -- and it's slow too, as it has to do all sorts of database queries for each order that comes in.

If you had written this using a rule engine, though, you'd have nice clean code, with one rule for each pricing policy. If somebody needed to know where the "Wacky Wednesday" pricing policy is implemented, it would be easy to find it.

And if that rule engine is a Jess-like system, it's not slow, either; the rule engine itself indexes all the orders and no lookups are required; the rule engine just "knows" what it needs to know about past orders.

11.3. Doing it with Jess

So without further ado, we'll implement the above idea with Jess. We're going to write an explicit "pricing library" which can be invoked from a Web app or from any other kind of Java code. There will be both a Java component and a Jess language component to what we'll write; we'll look at the Java part first. Note that all the code for this example, including a test driver, comes with Jess; it's in the directory `Jess71p2/examples/pricing_engine`. There's a `README` file there with some information about building and running the example.

Your Java code needs to create an instance of the Jess rule engine, load in the catalog data, then load in the rules (which we'll be writing later in this chapter.) This one instance of Jess can then be reused to process each order. (You only have to load the catalog data once; Jess will index it and later accesses will be fast.) We'll package all this up in a class named `PricingEngine`:

```
public class PricingEngine {
    private Rete engine;
    private WorkingMemoryMarker marker;
    private Database database;

    public PricingEngine(Database aDatabase) throws JessException {
        // Create a Jess rule engine
        engine = new Rete();
        engine.reset();

        // Load the pricing rules
        engine.batch("pricing.clp");

        // Load the catalog data into working memory
        database = aDatabase;
        engine.addAll(database.getCatalogItems());

       // Mark end of catalog data for later
       marker = engine.mark();
    }
```

(The `Database` interface is an application-specific data access wrapper; it's an interface and a trivial implementation is included in the example code.) Note that the call to `jess.Rete.batch(java.lang.String)` will find the file `myrules.clp` not only in the current directory but even if it's packaged in a jar file with the `PricingEngine` class, or put into the `WEB-INF/classes` directory of a Web application. Jess tries to find the file using a succession of different class loaders before giving up.

Then whenever you want to process an order, the pricing engine needs to do four things: reset the engine back to its initial state; load the order data; execute the rules; and extract the results. In this case the results will be `Offer` objects in working memory, put there by the rules; each `Offer` represents some kind of special pricing deal to be applied to the order. We'll write a short private routine for one of these steps, and then a single public method that performs all four steps on behalf of clients of the pricing engine.

First, a short routine to load the order data into Jess:

```
    private void loadOrderData(int orderNumber) throws JessException {
        // Retrive the order from the database
        Order order = database.getOrder(orderNumber);

        if (order != null) {
           // Add the order and its contents to working memory
           engine.add(order);
           engine.add(order.getCustomer());
           engine.addAll(order.getItems());
        }
    }
```

Now the pricing engine's business method, which takes an order number and returns an Iterator over the applicable offers. We use one of Jess's predefined `jess.Filter` implementations to select only the `Offer` objects from working memory.

```
    public Iterator run(int orderNumber) throws JessException {
        // Remove any previous order data, leaving only catalog data
        engine.resetToMark(marker);

        // Load data for this order
        loadOrderData(orderNumber);

        // Fire the rules that apply to this order
        engine.run();

        // Return the list of offers created by the rules
        return engine.getObjects(new Filter.ByClass(Offer.class));
    }
```

That's it! Now any servlet, EJB, or other Java code can instantiate a `PricingEngine` and use it to find the offers that apply to a given order... once we write the rules, that is.

A note about error handling: I've exposed JessException in the interface to all of these methods. I could have hidden JessException inside them by catching JessException and rethrowing a non-Jess exception. I've chosen not to do that here just to make the code shorter.

11.4. Making your own rules

All that's left is to write the rules. Our rules will be matching against some simple Java model objects. Imagine you've already got some model objects being used elsewhere in your system: Customer , Offer , Order , OrderItem , and CatalogItem .

Now all we have to do is express the business rules as Jess rules. Every rule has a name, an optional documentation string, some *patterns* , and some *actions.* A *pattern* is statement of something that must be true for the rule to apply. An *action* is something the rule should do if it does apply. Let's see how some of our pricing rules would look in Jess.

"Give a 10% discount to everybody who spends more than $100."

```
Jess> (defrule 10%-volume-discount
    "Give a 10% discount to everybody who spends more than $100."
    ?o <-(Order {total > 100})
    =>
    (add (new Offer "10% volume discount" (/ ?o.total 10))))
```

In a Jess rule, the patterns come before the "=>" symbol. This rule applies to `Order` with a `total` property greater than $100. Inside the curly braces, we can write expressions which look just like Boolean expressions in Java. All the properties of an object are available to these expressions just by using their property names. You can learn more about writing rule patterns here.

The actions come after the "=>" symbol. If this rule applies to an `Order`, a new Java `Offer` object is created with a value 10% of the order total, and that `Offer` is added to working memory using the `add` Our `PricingEngine` class will find these `Offer` objects later.

"Give a 25% discount on items the customer buys three or more of."

```
Jess> (defrule 25%-multi-item-discount
    "Give a 25% discount on items the customer buys three or more of."
    (OrderItem {quantity >= 3} (price ?price))
    =>
    (add (new Offer "25% multi-item discount" (/ ?price 4))))
```

In this rule, we use the value of the "price" property to compute the discount. Because we don't otherwise mention the "price" property, we include the simple expression "(price ?price)", which assigns the value of the property to the variable "?price."

"If somebody buys a CD writer, send them a free sample of CD-RW disks; but only if they're a repeat customer."

```
Jess> (defrule free-cd-rw-disks
    "If somebody buys a CD writer, send them a free sample of CD-RW
     disks, catalog number 782321; but only if they're a repeat customer.
    We use a regular expression to match the CD writer's description."
    (CatalogItem (partNumber ?partNumber) (description /CD Writer/))
    (CatalogItem (partNumber 782321) (price ?price))
    (OrderItem (partNumber ?partNumber))
    (Customer {orderCount > 1})
    =>
    (add (new Offer "Free CD-RW disks" ?price)))
```

This rule is more complicated because it includes *correlations* between objects. Because we used the same variable `?partNumber` in two different patterns, the rule will only match when the corresponding properties of two objects are equal -- i.e., when the `partNumber` of an `OrderItem` is the same as the `partNumber` of a `CatalogItem` that represents a CD writer.

Also noteworthy: we've tested the "description" property of the `CatalogItem` object with a simple Java *regular expression.* If the phrase "CD Writer" appears anywhere in the description, that CatalogItem will match the pattern.

11.5. Multiple Rule Engines

Each `jess.Rete` object represents an independent reasoning engine. A single program can include any number of engines. The individual `Rete` objects can each have their own working memories, agendas, and rulebases, and can all function in separate threads. You can use multiple identical engines in a pool, or each engine can have its own rules, perhaps because you intend for them to interact in some way.

11.5.1. Peering: Sharing Rules Among Multiple Engines

Of course, creating multiple independent engines and then loading the same set of rules into each seems inefficient: the same rules would be parsed and compiled once for each engine, and you'd have a lot of redundant data structures. Therefore Jess supports *peering* of rule engines. With peering, you load and compile a rule base just once to create the first engine, and

then you can create multiple independent *peer* engines which share the compiled rules. The peers all share their tables of rules, functions, templates, and defglobals, as well as their classloader, but each peer has its own agenda, working memory and execution context. That means they all apply the same set of rules, but to data unique to each peer. No peer sees facts asserted into the working memory of any other peer.

On the other hand, changes to the rule base, template list, function list, etc, are seen by all the members of a peered set, regardless of which engine the changes are made through.

All peers, including the first engine, are equivalent. Once a peer is created, the first engine and the peer behave identically; the first engine is not special in any way. Either of them can be used to create new peers.

Creating a peer is a fast operation, and of course peering uses less memory than creating independent engines. These properties make peers ideal for use in pools in Web applications. You can create a pool of peers, remove one from the pool to service a client request, and then return it to the pool.

The following simple example shows how to use peering. The API consists solely of a special constructor in the Rete class:

```
import jess.*;
public class ExPeering {
    public static void main(String[] argv) throws JessException {
        // Create the "original" engine
        Rete engine = new Rete();

        // Load a rule into it
        engine.eval("(defrule rule-1 (A ?x) => (printout t ?x crlf))");

        // Create a peer of the first engine
        Rete peer = engine.createPeer();

        // Assert different facts into the two engines
        engine.assertString("(A original)");
        peer.assertString("(A peer)");

        // Run the original engine; prints "original"
        engine.run();

        // Run the peer; prints "peer"
        peer.run();
    }
}
    C:\> java ExPeering


original
peer
```

11.6. Jess in a Multithreaded Environment

Jess can be used in a multithreaded environment. The `jess.Rete` class internally synchronizes itself using several synchronization locks. The most important lock is a lock on working memory: only one thread will be allowed to change the working memory of a given `jess.Rete` object at a time.

The `Rete.run()` method, like the `(run)` function in the Jess programming language, returns as soon as there are no more applicable rules. In a multithreaded environment, it is generally appropriate for the engine to simply wait instead, because rules may be activated due to working memory changes that happen on another thread. That's the purpose of the `Rete.runUntilHalt()` method and the `(run-until-halt)` function, which use Java's `wait()/notify()` system to pause the calling thread until active rules are available to fire. `runUntilHalt` and `(run-until-halt)` won't return until `Rete.halt()` or `(halt)` are called, as the names suggest.

Our `PricingEngine` class is stateful and not meant to be used in a multithreaded way. This is by design. If you want to use a `PricingEngine` in a Web application, then your choices are analagous to your choices in using JDBC `Connection` objects: either create new ones as needed, which is simple, but inefficient; or use an object pool and a finite collection of objects which are checked out, used, and returned to the pool. The important point is that multiple `PricingEngine` objects will be absolutely independent, and you can create as few or as many as you need for a given application.

11.7. Error Reporting and Debugging

I'm constantly trying to improve Jess's error reporting, but it is still not perfect. When you get an error from Jess (during parsing or at runtime) it is generally delivered as a Java exception. The exception will contain an explanation of the problem. If you print a stack trace of the exception, it can also help you understand what went wrong. For this reason, it is *very important* that, if you're embedding Jess in a Java application, you *don't* write code like this:

```
// Don't ignore exceptions like this!
try {
Rete engine = new Rete();
engine.eval("(gibberish!)");
} catch (JessException ex) { /* ignore errors */ }
```

If you ignore the Java exceptions, you will miss Jess's explanations of what's wrong with your code. Don't laugh - more people code this way than you'd think!
Anyway, as an example, if you attempt to load the folowing rule in the standard Jess command-line executable,

```
; There is an error in this rule
Jess> (defrule foo-1
(foo bar)
->
(printout "Found Foo Bar" crlf))
```

You'll get the following printout:

```
Jess reported an error in routine Jesp.parseDefrule.
```

```
Message: Expected '=>' at token '->'.
Program text: ( defrule foo-1 ( foo bar ) ->  at line 3.
```

This exception, like all exceptions reported by Jess, lists a Java routine name. The name `parseDefrule` makes it fairly clear that a rule was being parsed, and the detail message explains that `->` was found in the input instead of the expected `=>` symbol (we accidentally typed `->` instead). This particular error message, then, was fairly easy to understand.
Runtime errors can be more puzzling, but the printout will generally give you a lot of information. Here's a rule where we erroneously try to add the number `3.0` to the word `four`:

```
Jess> (defrule foo-2
=>
(printout t (+ 3.0 four) crlf))
```

This rule will compile fine, since the parser doesn't know that the `+` function won't accept the symbol `four` as an argument. When we `(reset)` and `(run)`, however, we'll see:

```
Jess reported an error in routine +
  while executing (+ 3.0 four)
  while executing (printout t (+ 3.0 four) crlf)
  while executing defrule MAIN::foo-2
  while executing (run).
Message: Not a number: four.
  Program text: ( run )  at line 12.
```

In this case, the error message is also pretty clear. It shows the offending function (+ 3.0 four); then the function that called that (printout); the message then shows the context in which the function was called (defrule MAIN::foo-2), and finally the function which caused the rule to fire (run).
The message 'Not a number: four' tells you that the `+` function wanted a numeric argument, but found the symbol `four` instead.
If we make a similar mistake on the LHS of a rule:

```
Jess> (defrule foo-3
(test (eq 3 (+ 2 one)))                                          -
=>
)
```

We see the following after a `reset`:

```
Jess reported an error in routine +
  while executing (+ 2 one)
  while executing (eq 3 (+ 2 one))
  while executing 'test' CE
  while executing rule LHS (TECT)
  while executing (reset).
Message: Not a number: one.
  Program text: ( reset )  at line 22.
```

Again, the error message is very detailed, and makes it clear, I hope, that the error occurred during rule LHS execution, in a `test` CE, in the function `(+ 2 one)`. Note that Jess cannot tell you which rule this LHS belongs to, since rule LHSs can be shared.

11.8. Creating Rules from Java

It is now possible to create Jess rules and queries using only the Java API -- i.e., without writing either a Jess language or XML version of the rule. This isn't recommended -- part of the power

of a rule engine is the ability it gives you to separate your rules from your other code -- but sometimes it may be worth doing.

Defining rules from Java is complex, and is still an undocumented process. If you're interested in doing it, your best resource is the source code for the `jess.xml` package, which uses Jess's public APIs to build rules. Be aware that these APIs may change without notice.

12. Adding Commands to Jess

The Java interface `jess.Userfunction` represents a single function in the Jess language. You can add new functions to the Jess language simply by writing a class that implements the `jess.Userfunction` interface (see below for details on how this is done), creating a single instance of this class and installing it into a `jess.Rete` object using `Rete.addUserfunction()`. The `Userfunction` classes you write can maintain state; therefore a `Userfunction` can cache results across invocations, maintain complex data structures, or keep references to external Java objects for callbacks. A single `Userfunction` can be a gateway into a complex Java subsystem.

12.1. Writing Extensions

I've made it as easy as possible to add user-defined functions to Jess. There is no system type-checking on arguments, so you don't need to tell the system about your arguments, and values are self-describing, so you don't need to tell the system what type you return. You do, however, need to understand several Jess classes, including `jess.Value`, `jess.Context`, and `jess.Funcall`, as previously discussed in the chapter Introduction to Programming with Jess in Java.

12.1.1. Implementing your Userfunction

To implement the `jess.Userfunction` interface, you need to implement only two methods: `getName()` and `call()`. Here's an example of a class called 'MyUpcase' that implements the Jess function `my-upcase`, which expects a String as an argument, and returns the string in uppercase.

```
import jess.*;

public class ExMyUpcase implements Userfunction {
    // The name method returns the name by which
    // the function will appear in Jess code.
    public String getName() { return "my-upcase"; }

    public Value call(ValueVector vv, Context context) throws JessException {
        String result = vv.get(1).stringValue(context).toUpperCase();
        return new Value(result, RU.STRING);
      }
  }
```

The `call()` method does the business of your Userfunction. When `call()` is invoked, the first argument will be a `ValueVector` representation of the Jess code that evoked your function. For example, if the following Jess function calls were made,

```
Jess> (load-function ExMyUpcase)

Jess> (my-upcase foo)

"FOO"
```

the first argument to `call()` would be a `ValueVector` of length two. The first element would be a `Value` containing the symbol (type `RU.SYMBOL`) `my-upcase`, and the second argument would be a `Value` containing the string (`RU.STRING`) `"foo"`.

Note that we use `vv.get(1).stringValue(context)` to get the first argument to `my-upcase` as a Java String. If the argument doesn't contain a string, or something convertible to a string, `stringValue()` will throw a `JessException` describing the problem; hence you don't need to worry about incorrect argument types if you don't want to. `vv.get(0)` will always return the symbol `my-upcase`, the name of the function being called (the clever programmer will note that this would let you construct multiple objects of the same class, implementing different functions based on the name of the function passed in as a constructor argument). If you want, you can check how many arguments your function was called with and throw a JessException if it was the wrong number by using the `vv.size()` method. In any case, our simple implementation extracts a single argument and uses the Java `toUpperCase()` method to do its work. `call()` must wrap its return value in a `jess.Value` object, specifying the type (here it is `RU.STRING`).

12.1.1.1. Legal return values

A `Userfunction` must return a valid `jess.Value` object; it cannot return the Java `null` value. To return "no value" to Jess, use `nil`. The value of `nil` is available in the public static final variable `jess.Funcall.NIL`.

12.1.2. Loading your Userfunction

Having written this class, you can then, in your Java main program, simply call `Rete.addUserfunction()` with an instance of your new class as an argument, and the function will be available from Jess code. So, we could have

```
import jess.*;
public class ExAddUF {
  public static void main(String[] argv) throws JessException {
    // Add the 'my-upcase' command to Jess
    Rete r = new Rete();
    r.addUserfunction(new ExMyUpcase());

    // This will print "FOO".
    System.out.println(r.eval("(my-upcase foo)"));
  }
}
C:\> java ExAddUF

"FOO"
```

Alternatively, the Jess language command `load-function` could be used to load `my-upcase` from Jess:

```
Jess> (load-function ExMyUpcase)

Jess> (my-upcase foo)

"FOO"
```

12.1.3. Calling assert from a Userfunction

The `jess.Rete.assertFact()` method has two overloads: one version takes a `jess.Context` argument, and the other does not. When writing a Userfunction, you should always use the first

version, passing the `jess.Context` argument to the Userfunction. If you do not, your Userfunction will not interact correctly with the `logical` conditional element.

12.2. Writing Extension Packages

The `jess.Userpackage` interface is a handy way to group a collection of `Userfunctions` together, so that you don't need to install them one by one (all of the extensions shipped with Jess are included in Userpackage classes). A `Userpackage` class should supply the one method `add()`, which should simply add a collection of `Userfunctions` to a `Rete` object using `addUserfunction()`. Nothing mysterious going on, but it's very convenient. As an example, suppose `MyUpcase` was only one of a number of similar functions you wrote. You could put them in a `Userpackage` class like this:

```
import jess.*;
public class ExMyStringFunctions implements Userpackage {
    public void add(Rete engine) {
        engine.addUserfunction(new ExMyUpcase());
        // Other similar statements
    }
}
```

Now in your Java code, you can call

```
import jess.*;
public class ExAddUP {
  public static void main(String[] argv) throws JessException {
    // Add the 'my-upcase' command to Jess
    Rete r = new Rete();
    r.addUserpackage(new ExMyStringFunctions());

    // This will still print FOO.
    System.out.println(r.eval("(my-upcase foo)"));
  }
}
C:\> java ExAddUP

"FOO"
```

or from your Jess code, you can call

```
Jess> (load-package ExMyStringFunctions)

Jess> (my-upcase foo)

"FOO"
```

to load these functions in. After either of these snippets, Jess language code could call `my-upcase`, `my-downcase`, etc.

`Userpackage`s are a great place to assemble a collection of interrelated functions which potentially can share data or maintain references to other function objects. You can also use `Userpackage`s to make sure that your `Userfunction`s are constructed with the correct constructor arguments.

All of Jess's "built-in" functions are simply `Userfunctions`, albeit ones which have special access to Jess' innards. Most of them are automatically loaded by code in the `jess.Funcall` class. You can use these as examples for writing your own Jess extensions.

12.3. Obtaining References to Userfunction Objects

Occasionally it is useful to be able to obtain a reference to an installed `Userfunction` object. The method `Userfunction Rete.findUserfunction(String name)` lets you do this easily. It returns the `Userfunction` object registered under the given name, or null if there is none. This is useful when you write Userfunctions which themselves maintain state of some kind, and you need access to that state.

13. Creating Graphical User Interfaces in the Jess Language

Jess, being just a set of Java classes, is easily incorporated as a library into graphical applications written in Java. It is also possible, though, to write graphical applications in the Jess language itself. The details of this are outlined in this chapter.

13.1. Handling Java AWT events

It should now be obvious that you can easily construct GUI objects from Jess. For example, here is a `Button`:

```
Jess> (defglobal ?*b* = (new java.awt.Button "Hello"))
```

What might not be obvious is how, from Jess, you can arrange to have something happen when the button is pressed. This section of this manual shows how to implement an interface using only Jess code. Therefore, all you need to do to create event handlers in Jess is to implement the appropriate interface.

An example should clarify matters. Let's say that when a `Hello` button is pressed, you would like the string `Hello, World!` to be printed to standard output (how original!). Here's a complete program in Jess that demonstrates how to do it:

```
Jess> (import java.awt.*)

Jess> (import java.awt.event.*)

Jess> ;; Create the widgets
(defglobal ?*f* = (new Frame "Button Demo"))

Jess> (defglobal ?*b* = (new Button "Hello"))

Jess> ;; Add a listener to the button
(?*b* addActionListener (implement ActionListener using (lambda (?name ?evt)
    (printout t "Hello, World!" crlf))))

Jess> ;; Assemble and display the GUI
(?*f* add ?*b*)

Jess> (?*f* pack)

Jess> (set ?*f* visible TRUE)
```

You'll have to quit this program using ^C. To fix this, you can add a `java.awt.event.WindowListener` which handles `WINDOW_CLOSING` events to the above program:

```
Jess>  ;; Add a WINDOW_CLOSING listener
(import java.awt.event.WindowEvent)
(?*f* addWindowListener  (implement WindowListener using (lambda (?name ?event)
    (if (= (?event getID) (WindowEvent.WINDOW_CLOSING)) then
        (exit)))))
```

Now when you close the window Jess will exit. Notice how we can examine the `?event` parameter for event information.

Because we imported the WindowEvent class, we were able to use "`(WindowEvent.WINDOW_CLOSING)`" to access this static data member. See the documentation for the `import` function for details. This technique is very often useful when programming GUIs with Jess.
We have used the "raw" AWT widgets here, but this same technique works fine with Swing as well.

```
Jess> (import javax.swing.*)

Jess> (import java.awt.event.*)

Jess> (import javax.swing.JFrame)

Jess> (defglobal ?*f* = (new JFrame "Button Demo"))

Jess> (defglobal ?*b* = (new JButton "Hello"))

Jess> (defglobal ?*p* = (get ?*f* contentPane))

Jess> (?*b* addActionListener (implement ActionListener using (lambda (?name ?event)
     (printout t "Hello, World!" crlf))))

Jess> (?*p* add ?*b*)

Jess> (?*f* pack)

Jess> (set ?*f* visible TRUE)

Jess> (?*f* setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE))
```

See the demo `examples/jess/jframe.clp` for a slightly more complex example of how you can build an entire Java graphical interface from within Jess.

13.2. Screen Painting and Graphics

As you may know, the most common method of drawing pictures in Java is to subclass either `java.awt.Canvas` or `javax.swing.JPanel`, overriding the `void paint(Graphics g)` or `void paintComponent(Graphics g)` method, respectively, to call the methods of the `java.awt.Graphics` argument to do the drawing. Well, Jess can't help you to subclass a Java class (at least not yet!), but it does provide adaptor classes, much like the event adaptors described above, that will help you draw pictures. The classes are named `jess.awt.Canvas` and `jess.swing.JPanel`. They can be used as normal Java GUI components: `Canvas` in AWT applications, and `JPanel` with Swing. When you construct an instance of these classes, you pass in the name of a Jess function and a reference to a `jess.Rete` object. Whenever the Java painting method is called to render the Jess component, the Jess component in turn will call the given function. The function will be passed two arguments: the or `jess.swing.JPanel` instance itself, and the `java.awt.Graphics` argument. In this way, Jess code can draw pictures using Java calls. An example looks like this:

```
Jess> ;; A painting deffunction. This function draws a red 'X' between the
;; four corners of the Canvas on a blue field.
(import java.awt.Color)
(deffunction painter (?canvas ?graph)
  (bind ?x (get-member (?canvas getSize) width))
  (bind ?y (get-member (?canvas getSize) height))
  (?graph setColor (Color.blue))
  (?graph fillRect 0 0 ?x ?y)
```

```
  (?graph setColor (Color.red))
  (?graph drawLine 0 0 ?x ?y)
  (?graph drawLine ?x 0 0 ?y))

Jess> ;; Create a canvas and install the paint routine.
(bind ?c (new jess.awt.Canvas painter (engine)))
```

Complete programs built on this example are in the files `examples/jess/awtdraw.clp` and `examples/jess/swingdraw.clp` in the Jess distribution.

14. Jess and XML

14.1. Introduction

14.1. Introduction

Jess now supports an XML-based rule language. Rather than adopting an evolving standard like RuleML, Jess's XML rule language (*JessML*) was designed to be close in structure to the Jess language itself. JessML is a hybrid imperative/declarative language, just like the Jess language itself. That means you can not only write rules in it, but define and call functions as well. Anything you can do in the Jess language, you can do in JessML too.

Transformation between JessML and RuleML and other XML rule languages should be simple using XSLT.

JessML code is quite verbose. Remember, though, that it's intended to be easy for machines, not humans, to work with.

14.2. The JessML Language

Jess comes with an XML schema for JessML; it's provided as the file `JessML1_0.xsd` in the `lib` subdirectory of the Jess distribution. The schema as written accepts all well-formed JessML code, but does not reject all invalid code; it does not include all the semantics of the JessML language (for example, the requirement that a template be defined before it is used in a pattern.) The JessML namespace URI is `http://www.jessrules.com/JessML/1.0` and should be included in all JessML documents. A well-formed JessML document should therefore be of the form

```
<?xml version='1.0' encoding='US-ASCII'?>
<rulebase xmlns='http://www.jessrules.com/JessML/1.0'>
...
</rulebase>
```

14.2.1. The rulebase Element

The top-level element in an executable JessML file is the *rulebase* element. A rulebase element can contain any other JessML elements in any order.

14.2.2. Names

Many constructs -- rules, templates, modules, functions -- have a name. In JessML, names are represented by a *name* element. A name element contains the text of the name, optionally including a module name:

- PLANNING::store-plan-data
- MAIN::animal
- make-pair

14.2.3. Values

Just as in Jess, simple data items in JessML are called *values*. A JessML value consists of a type attribute and the data itself. The type attribute can contain one of the type names in the jess.RU class. Here are a few examples of valid values:

- <value type="SYMBOL">nil</value>
- <value type="INTEGER">1</value>
- <value type="STRING">A String</value>

14.2.4. Function calls

The actions of rules, some of their tests and the bodies of deffunctions are made up of *funcalls*, short for "function calls." A JessML *funcall* tag contains a name element and a series of value elements as the arguments. The Jess function call "(+ ?x 1)" looks like this in JessML:

```
<funcall>
  <name>+</name>
  <value type="VARIABLE">x</value>
  <value type="INTEGER">1</value>
</funcall>
```

14.2.4.1. Special functions

A few special functions have slightly different syntax. The `modify` and `duplicate` functions each accept a single value plus one or more slot elements as arguments, while the `assert` function accepts one or more fact elements.

14.2.5. Slots

A *slot* in JessML is just the same as it is in Jess: it's a list (usually a two-element list, but sometimes longer) with a symbol as the head. In JessML, a slot element always starts with a name element, and contains one or more value elements; for example,

```
<slot>
    <name>location</name>
    <value type="SYMBOL">nil</value>
</slot>
```

14.2.6. Templates

You can define a template in the JessML language using the *template* element. A template element contains a name element followed by any number of elements. The value elements of the slots are interpreted as default values.

```
<template>
    <name>monkey</name>
    <slot>
        <name>species</name>
        <value type="SYMBOL">rhesus</value>
    </slot>
```

```
</template>
```

You can also automatically define a template based on a Java class using a *from-class* property, like this:

```
<template>
  <name>MAIN::button</name>
  <properties>
    <property>
      <name>from-class</name>
      <value type="SYMBOL">java.awt.Button</value></property>
  </properties>
</template>
```

14.2.7. Rules

A rule contains a name element, a *lhs* element, and a *rhs* element. The rhs element contains a collection of funcall elements representing the rule's actions. The lhs element contains groups and patterns, which are somewhat complex, and will be discussed in the following sections. A simple rule might look like this (this rule prints "Fired!" for each "MAIN::button" fact that exists:)

```
<rule>
    <name>MAIN::myrule</name>
    <lhs>
        <group>
            <name>and</name>
            <pattern>
                <name>MAIN::button</name>
            </pattern>
        </group>
    </lhs>
    <rhs>
        <funcall>
            <name>printout</name>
            <value type="SYMBOL">t</value>
            <value type="STRING">Fired!</value>
            <value type="SYMBOL">crlf</value>
        </funcall>
    </rhs>
</rule>
```

14.2.7.1. Groups

A *group* element represents one of Jess's grouping conditional elements: `and`, `or`, or `not`. Every rule has an "and" group enclosing all of its patterns, as in the example rule above. A group always contains a name element followed by any combination of nested groups or *pattern* elements.

14.2.7.2. Patterns

A pattern element represents a single pattern in a rule. It consists of a name element, an optional *binding* element, and any number of slot elements. The values of the slots should consist of *test* elements (described below.)

14.2.7.3. Pattern bindings

You can use a binding element to bind a variable to the fact that matches a pattern. The text of the binding element is simply the name of the variable to use.

14.2.7.4. Tests

A test element represents a single test in a pattern. A test can either be *discriminating* or *nondiscriminating.* A discriminating test is one that sometimes matches and sometimes doesn't. A nondiscriminating test is one that always matches; generally this refers to tests that merely bind a variable to the contents of a slot. There's no difference in the syntax between these two types of test.

A test element contains a *type* element, an optional *conjunction* element, and either a value element or a funcall element. The type element's body is either "eq" or "neq", corresponding to a positive or negative test. The conjunction element's body can be either "and" or "or", and it is can used only in the second and later tests on a slot. Finally, the value element represents the test itself. If it's a constant, the contents of the slot will be compared to it for equality or inequality, according to the contents of the type element. If the value is a variable, and if that variable has not been bound to a slot, the variable is bound to the current slot. Otherwise, the variable is compared to the contents of the slot. Finally, if there is a funcall element, the function is evaluated and the test passes if the result is true (for an "eq" test) or false (for a "neq" test.)

14.2.8. Fact files

The `save-facts-xml` function writes an XML-formatted file containing only Jess facts. The `load-facts` function knows how to read those files; the facts contained in the file will be immediately asserted. Any deftemplates needed must have been previously defined or an error will occur. You can create this kind of file yourself, as well. Such a file should be a complete, well-formed XML file containing an XML declaration and a `fact-list` root element. Inside the document should be nothing but `fact` elements, each representing a single Jess fact.

14.3. Writing Constructs in JessML

You can translate a file of Jess language code into XML using Jess's `jess.xml.XMLPrinter` class; this is a good way to learn how Jess and JessML correspond. You can use this tool from the command line like this:

```
java -classpath jess.jar jess.xml.XMLPrinter myfile.clp > myfile.xml
```

The file "myfile.xml" will be a complete translation of all the code in "myfile.clp", including templates, rules, deffunctions, and top-level function calls.

The `XMLPrinter` class and its companion `jess.xml.XMLVisitor` can also be used to translate Jess code into XML programatically. You could use XMLPrinter to translate a file of Java code like this:

```

```

```
import jess.xml.XMLPrinter;
import jess.JessException;
import java.io.*;
public class ExXMLPrinter {
    public static void main(String[] args) throws JessException, IOException {
        FileReader fr = new FileReader(args[0]);
        XMLPrinter printer = new XMLPrinter(new PrintWriter(System.out, true));
        try {
            printer.printFrontMatter();
            printer.translateToXML(fr);
            printer.printBackMatter();
        } finally {
            printer.close();
        }
    }
}
```

To translate individual constructs and other Jess objects that you've already read into Jess, you can use the `XMLVisitor` class. To write a single Jess object as XML, you create an `XMLVisitor` instance, passing the object as the constructor argument; any of the library classes that implement `jess.Visitable` will do. Then you can just call `jess.xml.XMLVisitor.toString()` on this instance to get the XML version.

```
import jess.xml.*;
import jess.*;
import java.io.*;
public class ExXMLVisitor {
    public static void main(String[] argv) throws JessException {
        Rete engine = new Rete();
        engine.eval("(defrule hello => )");
        XMLVisitor visitor = new XMLVisitor(engine.findDefrule("hello"));
        System.out.println(visitor);
    }
}
```

Jess comes with an XML Schema which you can use to validate the XML you write; it's the file `JessML1_0.xsd` in the `lib` subdirectory. Jess won't actually validate your code against this schema, as the standard JDK XML parser is a non-validating parser. To validate your code you'll need a validating parser like Apache Xerces.

14.4. Parsing JessML

In general, anywhere you can use a file full of Jess language code, you can also use a JessML file. For example, the file you provide as a command-line argument to `jess.Main` can be a JessML file.

The standard `batch` function can read JessML code as well as Jess language code. It decides what kind of file it is looking at by looking for the "<?xml" characters that should begin a well-formed XML file. `require`, which uses batch internally, will work with JessML as well.

You can get a little bit more control of the process by using the classes JessSAXParser or JessSAXHandler directly. JessSAXParser will parse a JessML document from an InputSource,

installing the results into a Rete object you provide. JessSAXHandler is a SAX event handler that you can use with your own parser.

15. The javax.rules API

15.1. Introduction

`javax.rules`, also known as JSR-94, is a Java runtime api for rule engines. It provides a very simple model of interaction with the rule engine itself, and a somewhat more involved mechanism for administering sets of rules.

The `javax.rules` API does *not* define a rule language of its own, but rather just an API for working with vendor-specific rule languages. In theory, you can write a program to the `javax.rules` API, and use it with different rule engines without modifying or recompiling the Java code. This would be accomplished by changing parameters in a configuration file, and swapping out one vendor-specific rule definition file for another. In practice, the `javax.rules` API fails to expose many of the interesting features of Jess directly, so this implementation provides a "hook" that will allow you to add Jess-specific code to take advantage of these features.

Finally, note that the reference implementation of `javax.rules` took the form of a driver for Jess. This driver was defined in the package `org.jcp.jsr94.jess` and used its own XML "wrapper" format for Jess language code. Jess 7 now includes its own `javax.rules` driver, defined in the package `jess.jsr94`. This new driver supports code written in the Jess language itself as well as JessML; the reference implementation's "wrapper" format is not supported. Please don't use the reference implementation driver; use the new `jess.jsr94` driver instead.

15.2. Using javax.rules

Using `javax.rules` to work with Jess is a two-stage process. In the first stage, you create and register a *rule execution set,* which is basically an instance of `jess.Rete` with templates, rules, and other necessary code already loaded. In the second stage, you retrieve the rule execution set from the registrar, load Java objects into it, and execute the rules.

15.2.1. Registering a RuleExecutionSet

Our first example shows how to create a rule execution set from a file of Jess code and how to register it with `javax.rules`. You could use an XML file instead of a Jess file; the code would not change. Exception handling has been left out for brevity. Most of the methods in the `javax.rules` API throw both a subclass of `javax.rules.RuleException` and `java.rmi.RemoteException`.

Typically, a `RuleExecutionSet` is created once and executed multiple times. The intention behind the `javax.rules` design is that `RuleExecutionSet`s would be created using an app server's configuration tools, and then used by application code.

```
import javax.rules.*;
import javax.rules.admin.*;
import java.util.HashMap;
import java.io.FileReader;
```

```
public class ExJSR94Register {
  public static final String URI = "rules://test-rules";
  public static final String RULE_SERVICE_PROVIDER = "jess.jsr94";

  public static void main(String[] argv) throws Exception {
    // Load the rule service provider.
    // Loading this class will automatically register this
    // provider with the provider manager.
    Class.forName(RULE_SERVICE_PROVIDER + ".RuleServiceProviderImpl");

    // Get the rule service provider from the provider manager.
    RuleServiceProvider serviceProvider =
        RuleServiceProviderManager.getRuleServiceProvider(RULE_SERVICE_PROVIDER);

    // get the RuleAdministrator
    RuleAdministrator ruleAdministrator =
        serviceProvider.getRuleAdministrator();

    // get an input stream to a ruleset
    FileReader reader = new FileReader("rules.clp");

    // parse the ruleset
    try {
        // "properties" holds vendor-specific information
        // Jess doesn't use this parameter for these methods
        HashMap properties = new HashMap();

        // Create the RuleExecutionSet
        RuleExecutionSet executionSet =
            ruleAdministrator.getLocalRuleExecutionSetProvider(properties).
            createRuleExecutionSet(reader, properties);

        // register the RuleExecutionSet
        ruleAdministrator.registerRuleExecutionSet(URI,
                                                   executionSet, properties);

    } finally {
        reader.close();
    }
  }
}
```

15.2.2. Using a RuleExecutionSet

Now that we have registered a `RuleExecutionSet`, we can use it to run the rules. For this example, let's use the following trivial rule file `rules.clp`:

```
Jess> (defrule foo
=>
(add (new String "Hello, World!")))
```

When executed, this rule will fire and add a single String to working memory. Here is the Java code to retrieve and execute the `RuleExecutionSet`:

```
import javax.rules.*;
import javax.rules.admin.*;
```

```
import java.util.*;

public class ExExecuteJSR94 {
  public static final String URI = "rules://test-rules";
  public static final String RULE_SERVICE_PROVIDER = "jess.jsr94";

  public static void main(String[] argv) throws Exception {
    // Get the rule service provider from the provider manager.
    RuleServiceProvider serviceProvider =
        RuleServiceProviderManager.getRuleServiceProvider(RULE_SERVICE_PROVIDER);

    // Get a RuleRuntime
    RuleRuntime runtime = serviceProvider.getRuleRuntime();

    // Create a StatelessRuleSession
    StatelessRuleSession session = (StatelessRuleSession)
        runtime.createRuleSession(URI, new HashMap(),
                                  RuleRuntime.STATELESS_SESSION_TYPE);

    // Create a input list.
    List input = new ArrayList();

    // Execute the rules
    List results = session.executeRules(input);

    // Release the session.
    session.release();

    // Report the results
    System.out.println(results);
  }
}
C:\> java ExExecuteJSR94
```

When you run this program, if all goes well it will print "[Hello, World!]," displaying the single object in working memory.

15.2.3. Working with Java Objects

When we called `executeRules()` in the example above, we passed an empty `List` as an argument. You can instead pass in a `List` populated with Java objects. Jess will, for each object:

1. Determine the name of the class, without the package name.
2. Ensure that a template has been created under that name, or create a new one.
3. Add that object to working memory under that class name.

Jess will then execute the rules. You therefore should write your rules assuming that the objects will be filed in working memory under their class name with the package prefix removed. Note that your rule file should define appropriate templates for any objects actually matched by your rules using `deftemplate` or `defclass`, or the rules won't parse.

In this release, there's no way to control how these templates will be placed into modules, and there's no way to express Jess's concept of inheritance. These features will be added in a later release.

15.2.4. Going Further

There's a lot more to the `javax.rules` API than we've covered here. In particular, besides stateless rule sessions, there are stateful rule sessions that let you add and remove objects over time. Furthermore, there are many different ways to create a RuleExecutionSet. Besides creating one from a local file, they can be created from XML DOM trees and directly from existing `jess.Rete` objects. Interested readers are directed to the specification itself for more information until more information is added to this manual.

16. The Jess Function List

In this chapter, every Jess language function shipped with Jess version 7 is described. Previous versions of Jess had a notion of "optional functions," but Jess 7 does not: all of the functions described here are always available.
You may also like to look at a list of functions grouped by category or sorted by name.
Note: many functions documented as requiring a specific minimum number of arguments will actually return sensible results with fewer; for example, the + function will return the value of a single argument as its result. This behavior is to be regarded as undocumented and unsupported. In addition, all functions documented as requiring a specific number of arguments will not report an error if invoked with more than that number; extra arguments are simply ignored.

16.1. (- <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions
Returns:
Number
Description:
Returns the first argument minus all subsequent arguments. The return value is an INTEGER or LONG unless any of the arguments are FLOAT, in which case it is a FLOAT.

16.2. (-- <variable>)

Arguments:
A variable
Returns:
Number
Description:
Subtracts one from the variable (which should contain a numeric value,) sets the variable to the new value, and returns the new value. Throws an exception if the argument is not a variable containing a numeric type. The type of the variable is preserved.

16.3. (/ <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions
Returns:
Number
Description:
Returns the first argument divided by all subsequent arguments. The return value is a FLOAT.

16.4. (* <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions
Returns:
Number
Description:

Returns the products of its arguments. The return value is an `INTEGER` or `LONG` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

16.5. (** <numeric-expression> <numeric-expression>)

Arguments:
Two numeric expressions
Returns:
Number
Description:
Raises its first argument to the power of its second argument (using Java's `Math.pow()` function). ***Note*:** the return value may be `NaN` (not a number); see the Java API documentation for details.

16.6. (+ <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions
Returns:
Number
Description:
Returns the sum of its arguments. The return value is an `INTEGER` or `LONG` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

16.7. (++ <variable>)

Arguments:
A variable
Returns:
Number
Description:
Adds one to the variable (which should contain a numeric value,) sets the variable to the new value, and returns the new value. Throws an exception if the argument is not a variable containing a numeric type. The type of the variable is preserved.

16.8. (< <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions or `Comparable` objects
Returns:
Boolean
Description:
Returns `TRUE` if each argument is less than the argument following it; otherwise, returns `FALSE`. The definition of "less than" varies depending on the types being compared.

16.9. (<= <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions or `Comparable` objects
Returns:
Boolean
Description:

Returns TRUE if the value of each argument is less than or equal to the value of the argument following it; otherwise, returns FALSE. The definition of "less than or equal to" depends on the objects being compared.

16.10. (<> <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions or `Comparable` objects
Returns:
Boolean
Description:
Returns TRUE if the value of the first argument is not equal in value to all subsequent arguments; otherwise returns FALSE. The definition of "not equal" depends on the objects being compared.

16.11. (= <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions or `Comparable` objects
Returns:
Boolean
Description:
Returns TRUE if the value of the first argument is equal in value to all subsequent arguments; otherwise, returns FALSE. The integer 2 and the float 2.0 are =, but not eq.

16.12. (> <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions or `Comparable` objects
Returns:
Boolean
Description:
Returns TRUE if the value of each argument is greater than that of the argument following it; otherwise, returns FALSE. The definition of "greater than" depends on the objects being compared

16.13. (>= <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions or `Comparable` objects
Returns:
Boolean
Description:
Returns TRUE if the value of each argument is greater than or equal to that of the argument following it; otherwise, returns FALSE. The definition of "greater than or equal to" depends on the objects being compared.

16.14. (abs <numeric-expression>)

Arguments:
One numeric expression
Returns:
Number
Description:
Returns the absolute value of its only argument.

16.15. (add <Java object>)

Arguments:
A Java object
Returns:
nil
Description:
Adds the given object to working memory. Creates a "shadow fact" representing the given Java object, using the template whose name is the same as the given object's class, without the package prefix. If this template doesn't exist, it is created. Equivalent to calling

```
(definstance <classname> <Java object> auto)
```

See the description of the `definstance` function for more information.

16.16. (agenda [<module-name> | *])

Arguments:
Optionally, a module name or the symbol "*"
Returns:
NIL
Description:
Displays a list of rule activations to the WSTDOUT router. If no argument is specified, the activations in the current module (not the focus module) are displayed. If a module name is specified, only the activations in that module are displayed. If "*" is specified, then all activations are displayed.

16.17. (and <expression>+)

Arguments:
One or more expressions
Returns:
Boolean
Description:
Returns TRUE if all arguments evaluate to a non-FALSE value; otherwise, returns FALSE.

16.18. (apply <expression>+)

Arguments:
A function name or lambda followed by zero or more expressions
Returns:
An expression
Description:
Returns the result of calling the first argument, either the name of a Jess function or a lambda expression, on all the remaining arguments. The strength of this method lies in the fact that you can call a function whose name, for instance, is in a Jess variable.

16.19. (asc <string>)

Arguments:
Any string or symbol
Returns:
Integer
Description:
Returns the Unicode value of the first character of the argument, as an RU.INTEGER.

16.20. (as-list <java-object>)

Arguments:
A Java object (must be an array)
Returns:
A list
Description:
Converts the given Java array into a Jess list. This happens automatically for arrays returned by functions declared to return an array, but you can easily fetch an array from a Java method that returns Object, and in this case this function is useful.

16.21. (assert <fact>+)

Arguments:
One or more facts
Returns:
A Fact, or FALSE
Description:

Adds all the facts to the working memory; returns the last fact asserted or `FALSE` if no facts were successfully asserted (for example, if all facts given are duplicates of existing facts.) A `jess.JessEvent` of type `JessEvent.FACT` will be sent if the event mask is set appropriately.

It is important to remember that all pattern-matching happens during calls to `assert`, `retract`, `modify`, and related functions. Pattern-matching happens whether the engine is running or not.

Example:

```
Jess> (reset)
TRUE
Jess> (assert (testing 1 2 3))
<Fact-1>
```

16.22. (assert-string <string-expression>)

Arguments:
One string representing a fact
Returns:
A Fact, or FALSE
Description:

Converts a string into a fact and asserts it. Attempts to parse string as a fact, and if successful, returns the value returned by assert with the same fact. Note that the string must contain the fact's enclosing parentheses.

Example:

```
Jess> (reset)
TRUE
```

```
Jess> (assert-string "(testing 1 2 3)")

<Fact-1>
```

16.23. (bag <bag-command> <bag-arguments>+)

Arguments:
A symbol (a sub-command) and one or more additional arguments
Returns:
(Varies)
Description:
The `bag` command lets you manipulate Java hashtables from Jess. The net result is that you can create any number of associative arrays or property lists. Each such array or list has a name by which it can be looked up. The lists can contain other lists as properties, or any other Jess data type.
The `bag` command does different things based on its first argument. It's really seven commands in one:

- `create` accepts a String, the name of a new Bag to be created. The bag object itself is returned. For example:
-
```
Jess> (bag create my-bag)
```

- `delete` accepts the name of an existing bag, and deletes it from the list of bags.
- `find` accepts the name of a bag, and returns the corresponding bag object, if one exists, or `nil`.
- `list` returns a list of the names of all the existing bags.
- `set` accepts as arguments a bag, a String property name, and any Jess value as its three arguments. The named property of the given bag is set to the value, and the value is returned.
- `get` accepts as arguments a bag and a String property name. The named property is retrieved and returned, or `nil` if there is no such property. For example:
-
```
Jess> (defglobal ?*bag* = 0)

TRUE

Jess> (bind ?*bag* (bag create my-bag))

<Java-Object:java.util.Hashtable>

Jess> (bag set ?*bag* my-prop 3.0)

3.0

Jess> (bag get ?*bag* my-prop)

3.0
```

- `props` accepts a bag as the single argument and returns a list of the names of all the properties of that bag.

16.24. (batch <filename> [<charset>])

Arguments:
One string representing the name of a file, and optionally a character set name
Returns:

(Varies)

Description:

Attempts to parse and evaluate the given file as Jess code. If successful, returns the return value of the last expression in the file.

Note: the argument must follow Jess' rules for valid strings. On UNIX systems, this presents no particular problems, but Win32 filenames may need special treatment. In particular: pathnames should use either '\\' (double backslash) or '/' (forward slash) instead of '\' (single backslash) as directory separators.
In an applet, batch will try to find the file relative to the applet's document base. In any program, if the file is not found, the name is then passed to `ClassLoader.getSystemResourceAsStream()`. This allows files along the class path, including files in JARs, to be batched.

If you specify a character set name as the optional second argument, Jess will assume the file is written using that character set.

16.25. (bind <variable> <expression>)

Arguments:

A variable name and any value

Returns:

(Varies)

Description:

Binds a variable to a new value. Assigns the given value to the given variable, creating the variable if necessary. Returns the given value.

Example:

```
Jess> (bind ?x 3)
3
Jess> ?x
3
```

If the variable name contains a period, like `?x.y`, then this function will attempt to modify slot y of a fact in variable ?x to the given value.

16.26. (bit-and <integer-expression>+)

Arguments:

One or more integer expressions

Returns:

int

Description:

Performs the bitwise AND of the arguments. `(bit-and 7 4)` is 4, and is equivalent to the Java `7 & 4`.

16.27. (bit-not <integer-expression>)

Arguments:
One integer expression
Returns:
int
Description:
Performs the bitwise NOT of the argument. `(bit-not 0)` is -1, and is equivalent to the Java `~0`.

16.28. (bit-or <integer-expression>+)

Arguments:
One or more integer expressions
Returns:
int
Description:
Performs the bitwise OR of the arguments. `(bit-or 2 4)` is 6, and is equivalent to the Java `2 | 4`.

16.29. (bload <filename>)

Arguments:
One string representing the name of a file
Returns:
TRUE
Description:

The argument is the path to a file previously produced by the `bsave` command. The file is decompressed and deserialized to restore the state of the current Rete object. I/O routers are not restored from the file; they retain their previous state. Furthermore, JessListeners are not restored from the file; again, they are retained from their state prior to the bload.

To decode the dump file, this function decompresses the data using `java.util.zip.GZIPInputStream` and sends the result to the `jess.Rete.bload(java.io.InputStream)` method. Therefore, this function cannot be directly read data saved by `jess.Rete.bsave(java.io.OutputStream)` method because the decompression step will fail.

16.30. (break)

Arguments:
None
Returns:
N/A
Description:
Immediately exit any enclosing loop or control scope. Can be used inside of `for`, `while`, and `foreach` loops, as well as within the body of a `deffunction` or the right hand side of a `defrule`. If called anywhere else, will throw an exception.

16.31. (bsave <filename>)

Arguments:
One string representing the name of a file
Returns:

TRUE

Description:

Dumps the engine in which it is called to the given filename argument in a format that can be read using `bload`. Any input/output streams and event listeners are not saved during the serialization process.

To produce the dump file, this function calls the `jess.Rete.bsave(java.io.OutputStream)` method and compresses the data using `java.util.zip.GZIPOutputStream`. Therefore, data saved by this method cannot be directly read by the `jess.Rete.bload(java.io.InputStream)` method without first decompressing it.

16.32. (build <string-expression>)

Arguments:

One string representing some Jess code

Returns:

(Varies)

Description:

Evaluates a string as though it were entered at the command prompt. Only allows constructs to be evaluated. Attempts to parse and evaluate the given string as Jess code. If successful, returns the return value of the last expression in the string. This is typically used to define rules from Jess code. For instance:

```
(build "(defrule foo (foo) => (bar))")
```

Note: The string must consist of one single construct; multiple constructs can be built using multiple calls to `build`.

16.33. ([call] <java object> | <class-name> <method-name> <argument>*)

Arguments:

A java object or class name, optionally a method or field name, and any number of additional arguments

Returns:

(Varies)

Description:

Calls a Java method on the given object, a static method of the class named by the first argument, invokes a lambda, or returns the value of a Java field in the object. Unless the first argument is a lambda, the second argument is the name of the method or field. All subsequent arguments, if any, are passed to the lambda or method. When calling Java methods, arguments are promoted and overloaded methods selected precisely as for `new`. The return value is converted to a suitable Jess value before being returned. Array return values are converted to lists.

The functor `call` may be omitted unless the method being called is a Java static method. The following two method calls are equivalent:

```
Jess> (bind ?list (new java.util.ArrayList))

Jess> ;; These are legal and equivalent
(call ?list add "Foo")
(?list add "Foo")
```

Note that `call` can even be omitted if the object comes from the return value of another function call:

```
Jess> ;; This is legal
```

```
((new java.util.ArrayList 10) add "Foo")
```

If the first argument is a Java object, and the second a symbol, and there are no other arguments, and the object does not have a no-argument method by the given name, then Jess will attempt to return the value of a field by that name in the object, if one exists. As with method calls, the "call" functor can be omitted. So, for example,

```
Jess> (bind ?dim (new java.awt.Dimension 10 20))

Jess> ;; These are legal and equivalent
(get-member ?dim width)
(?dim width)
```

16.34. (call-on-engine <Java object> <jess-code>)

Arguments:
an `jess.Rete` object, and an executable snippet of Jess code
Returns:
(Varies)
Description:
Executes some Jess code in the context of the given Rete object. This is a nice way to send messages between multiple Rete engines in one process. Note that the current variable context is used to evaluate the code, so (for instance) all defglobal values will be from the calling engine, not the target.

16.35. (clear)

Arguments:
None
Returns:
TRUE
Description:
Clears Jess. Deletes all rules, deffacts, defglobals, templates, facts, activations, and so forth. Java Userfunctions are not deleted.

16.36. (clear-focus-stack)

Arguments:
None
Returns:
nil
Description:
Removes all modules from the focus stack.

16.37. (clear-storage)

Arguments:
None
Returns:
TRUE
Description:
Clears the hashtable used by `store` and `fetch`.

16.38. (close <router-identifier>*)

Arguments:
One or more router identifiers (symbols)
Returns:
TRUE
Description:
Closes any I/O routers associated with the given name by calling `close()` on the underlying stream, then removes the routers. Any I/O errors are ignored. Any subsequent attempt to use a closed router will report `bad router`. See open.

16.39. (complement$ <list-expression> <list-expression>)

Arguments:
Two lists
Returns:
List
Description:
Returns a new list consisting of all elements of the second list not appearing in the first list.

16.40. (context)

Arguments:
None
Returns:
A `jess.Context` object
Description:
Returns the execution context (a `jess.Context` object) it is called in. This provides a way for deffunctions to get a handle to this useful class.

16.41. (continue)

Arguments:
None
Returns:
N/A
Description:
Immediately jump to the end of any enclosing loop and begin the next iteration of the loop. For "while" loops, the loop test will be executed next; for "for" loops, the increment function will be executed. Can be used inside of `for`, `while`, and `foreach` loops. If called anywhere else, will throw an exception.

16.42. (count-query-results <query-name> <expression>*)

Arguments:
A query name, and zero or more additional expressions
Returns:
INTEGER
Description:
Runs a query and returns a count of the matches. See the documentation for defquery for more details. Also see run-query for caveats concerning calling count-query-results on a rule RHS.

16.43. (create$ <expression>*)

Arguments:
Zero or more expressions
Returns:
List
Description:
Creates and returns a new list containing all the given arguments, in order. For each argument that is a list, the individual elements of the list are added to the new list; this function will not create nested lists (which are not meaningful in the Jess language.) ***Note***: lists must be created explicitly using this function or others that return them. Lists cannot be directly parsed from Jess input.

16.44. (defadvice (before | after) (<function-name> | <list> | ALL) <function-call>+)

Arguments:
The symbol `before` or the symbol `after`, followed by either one function name or a list of function names or the symbol `ALL`, followed by one or more function calls.
Returns:
(varies)
Description:
Lets you supply extra code to run before or after the named function(s) or all functions. If `before` is specified, the code will execute before the named function(s); the variable $?argv will hold the entire function call vector (function name and parameters) on entry to and exit from the code block. If `after` is specified, the function will be called before the code block is entered. When the block is entered, the variable ?retval will refer to the original function's return value.
Whether `before` or `after` is specified, if the code block explicitly calls `return` with a value, the returned value will appear to the caller to be the return value of the original function. For `before` advice, this means the original function will not be called in this case.

16.45. (defclass <template-name> <Java class name> [extends <template-name>])

Arguments:
Two or four symbols, as noted above
Returns:
The second argument
Description:

Defines a template with the given name, with slots based on the Java Beans properties found in the named class. If the optional extends clause is included, the second named template will become the parent of the new template. The common slots in the two templates will be in the same order, at the beginning of the new template. Rules defined to match instances of the parent template will also match instances of the new child template.

Note that anything you can do using the defclass function, you can also do with the `deftemplate` construct -- but deftemplate lets you do even more.

16.46. (definstance <template-name> <Java object> [static | dynamic | auto])

Arguments:
A symbol, a Java object, and (optionally) one of the symbols `static`, `dynamic`, or `auto`
Returns:
The new shadow fact

Description:

Creates a "shadow fact" representing the given Java object, according to the named template. By default, or if the "dynamic" qualifier is specified, and if the object accepts `java.beans.PropertyChangeListener`s, then Jess will install a listener in the given object, so that Jess can keep the shadow fact updated if the object's properties change. If the object doesn't accept `PropertyChangeListener`s, or if the "static" qualifier is specified, then no listener will be registered and the shadow fact will not be upated when the object changes. The "auto" qualifier is equivalent to the default behavior.

Note that it is an error for a given Java object to be added to working memory more than once. The second and subsequent `definstance` calls for a given object will return a fact-id with value -1.

This function is a generalized form of `add`.

16.47. (delete$ <list-expression> <begin-integer-expression> <end-integer-expression>)

Arguments:
A list and two integer expressions
Returns:
List
Description:
Returns a new list like the original list but with the elements in the specified range removed. The first numeric expression is the 1-based index of the first element to remove; the second is the 1-based index of the last element to remove. It is an error to use any index value less than 1 or greater than the list length, or an end index less than the begin index.

16.48. (dependencies <fact-id>)

Arguments:
A Fact
Returns:
A list
Description:
Returns a list containing all the `jess.Token` objects that give logical support from the argument fact; if there are none this function returns an empty list. A `jess.Token` object is a list of facts; they're the same objects returned by `run-query`.

16.49. (dependents <fact-id>)

Arguments:
A Fact or fact-id
Returns:
A list
Description:
Returns a list containing all the `jess.Fact` objects that get logical support from the argument fact; if there are none this function returns an empty list.

16.50. (div <numeric-expression> <numeric-expression>+)

Arguments:
Two or more numeric expressions
Returns:

Number

Description:

Returns the first argument divided by all subsequent arguments using integer division. The return value is an `INTEGER`.

16.51. (do-backward-chaining <template-name>)

Arguments:

Name of a template (ordered or unordered)

Returns:

TRUE

Description:

Marks a template as being eligible for backwards chaining, as described in the text. If the template is unordered -- i.e., if it is explicitly defined with a (deftemplate) construct -- then it must be defined *before* calling `do-backward-chaining`. In addition, this function must be called *before* defining any rules which use the template.

16.52. (duplicate <fact-specifier> (<slot-name> <value>)+)

Arguments:

A fact and one or more two-element lists

Returns:

A fact

Description:

Makes a copy of the fact; the fact must be an unordered fact. Each list is taken as the name of a slot in this fact and a new value to assign to the slot. A new fact is asserted which is similar to the given fact but which has the specified slots replaced with new values. The new fact is returned. It is an error to call `duplicate` on a shadow fact.

As of Jess version 7, the slot names can be variables.

16.53. (e)

Arguments:

None

Returns:

Number

Description:

Returns the transcendental number *e*.

16.54. (engine)

Arguments:

None

Returns:

A `jess.Rete` object

Description:

Returns the `jess.Rete` object in which the function is called.

16.55. (eq <expression> <expression>+)

Arguments:
Two or more arbitrary arguments
Returns:
Boolean
Description:

Returns `TRUE` if the first argument is equal in type and value to all subsequent arguments. For strings, this means identical contents. Uses the Java `Object.equals()` function, so can be redefined. Note that the integer `2` and the floating-point number `2.0` are *not* `eq`, but they are `eq*` and `=`.

While often used in procedural code, this function is only rarely used during pattern matching. Direct matching is preferable.

16.56. (eq* <expression> <expression>+)

Arguments:
Two or more arbitrary arguments
Returns:
Boolean
Description:
Returns `TRUE` if the first argument is equivalent to all the others. Uses numeric equality for numeric types, unlike `eq`. Note that the integer 2 and the floating-point number 2.0 are *not* `eq`, but they are `eq*` and `=`.

16.57. (eval <lexeme-expression>)

Arguments:
One string containing a valid Jess expression
Returns:
(Varies)
Description:
Evaluates a string as though it were entered at a command prompt. Only allows functions to be evaluated. Evaluates the string as if entered at the command line and returns the result.
Note: The string must consist of one single function call; multiple calls can be evaluated using multiple calls to eval.

16.58. (evenp <expression>)

Arguments:
One numeric expression
Returns:
Boolean
Description:
Returns `TRUE` for even numbers; otherwise, returns `FALSE`. Results with non-integers may be unpredictable.

16.59. (exit)

Arguments:
None

Returns:
Nothing
Description:
Exits Jess and halts Java.

16.60. (exp <numeric-expression>)

Arguments:
One numeric expression
Returns:
Number
Description:
Raises the value *e* to the power of its only argument.

16.61. (explode$ <string-expression>)

Arguments:
One string
Returns:
List
Description:
Creates a list value from a string. Parses the string as if by a succession of `read` calls, then returns these individual values as the elements of a list.

16.62. (external-addressp <expression>)

Arguments:
One expression
Returns:
Boolean
Description:
Deprecated. Use `java-objectp` instead.

16.63. (fact-id <integer>)

Arguments:
One number, a fact-id
Returns:
The given number as a `jess.Fact`
Description:
If the argument is the fact-id of an existing fact, returns that `jess.Fact` object; otherwise throws an exception. The lookup done by this function is *slow*. Be sure you really need to call this function. If you have a value that prints as "<Fact-1>", then it's already a Fact object.

16.64. (facts [<module name> | *])

Arguments:
Module name or * (optional)
Returns:
TRUE
Description:

Prints a list of all facts in working memory from the current module. If a module name is provided as an argument, facts from that module are listed instead. If the argument "*" is given, all facts from all modules are listed.

16.65. (fact-slot-value <fact-id> <slot-name>)

Arguments:
A `jess.Fact` and a slot name
Returns:
(varies)
Description:
Returns the value in the named slot of the fact. You should *never* need to call this on the left hand side of a rule; direct matching is always better. You should only occasionally need it in other code; again, directly matching the slots on the left hand side of a rule is better.

16.66. (fetch <string or symbol>)

Arguments:
One string or symbol
Returns:
(varies)
Description:
Retrieves and returns any value previously stored by the `store` function under the given name, or `nil` if there is none. Analogous to the `fetch()` member function of the `Rete` class. See the section on using store and fetch for details.

16.67. (filter <predicate function> <list>)

Arguments:
A function and a list
Returns:
A list
Description:
Calls the function on each item in the list; returns a list of all the results for which the function does not return FALSE. The function can either be the name of a Userfunction, or it can be a `lambda` expression.

16.68. (first$ <list-expression>)

Arguments:
One list
Returns:
List
Description:
Returns the first field of a list as a new 1-element list.

16.69. (float <numeric-expression>)

Arguments:
One numeric expression
Returns:
Floating-point number
Description:

Converts its only argument to a float.

16.70. (floatp <expression>)

Arguments:
One numeric expression
Returns:
Boolean
Description:
Returns TRUE for floats; otherwise, returns FALSE.

16.71. (focus <module-name>+)

Arguments:
One or more symbols, the names of modules
Returns:
The name of the previous focus module
Description:
Changes the focus module. The next time the engine runs, the first rule to fire will be from the first module listed (if any rules are activated in this module.) The previously active module is pushed down on the focus stack. If more than one module is listed, they are pushed onto the focus stack in order from right to left.

16.72. (for <initializer> <condition> <increment> <body expression>*)

Arguments:
Three or more expressions
Returns:
(Varies)
Description:
Jess's `for` function works just like Java's `for` loop. First the initializer is evaluated. Then the condition is evaluated. If it does not evaluate to FALSE, the body expressions are evaluated in order. Next, the increment is evaluated, and then the condition is checked again. The loop continues until the condition evaluates to FALSE or until a `return` or `break` is encountered. Example:

```
Jess> (for (bind ?i 0) (< ?i 10) (++ ?i)
    (printout t ?i crlf))
```

In Java, the initializer, condition, or increment can be empty. In Jess, you can use the constant `nil` as an equivalent.

16.73. (foreach <variable> <list-expression> <action>*)

Arguments:
A variable, a list expression or iterator, and zero or more arguments
Returns:
Varies
Description:

The named variable is set to each value from the list, in turn. For each value, all of the other arguments are evaluated in order. The `break` function can be used to break the iteration.

Example:

```
Jess> (foreach ?x (create$ a b c d) (printout t ?x crlf))

a
b
c
d
```

There are two ways to specify the series of values: you can either specify a Jess list (as created by `create$`) or you can provide a `java.util.Iterator`.

16.74. (format <router-identifier> <string-expression> <expression>*)

Arguments:
A router identifier, a format string, and zero or more arguments
Returns:
A string
Description:
Sends formatted output to the specified logical name. Formats the arguments into a string according to the format string, which is identical to that used by `printf` in the C language (find a C book for more information). Returns the string, and optionally prints the string to the named router. If you pass `nil` for the router name, no printing is done.

16.75. (gensym*)

Arguments:
None
Returns:
Symbol
Description:
Returns a special unique sequenced value. Returns a unique symbol which consists of the letters `gen` plus an integer. Use `setgen` to set the value of the integer to be used by the next gensym call.

16.76. (get <Java object> <string-expression>)

Arguments:
A Java object and a string.
Returns:
(Varies)
Description:

Retrieves the value of a JavaBean's property or instance variable. The first argument is the JavaBean and the second argument is the name of the property or variable. Jess will first try to find a JavaBean property by this name; if none is found, it will look for an instance variable. The return value is converted to a suitable Jess value exactly as for call.

If applied to a `jess.Fact` object `get` returns the value of the named slot, exactly as for `fact-slot-value`.

16.77. (get-current-module)

Arguments:
None
Returns:

The name of the current module

Description:

Gets the current module (see set-current-module).

16.78. (get-focus)

Arguments:

None

Returns:

Symbol

Description:

Returns the name of the current focus module (see focus).

16.79. (get-focus-stack)

Arguments:

None

Returns:

List

Description:

Returns the module names on the focus stack as a list. The top module on the stack is the first entry in the list.

16.80. (get-member (<Java object> | <string-expression>) <string-expression>)

Arguments:

A Java object or a string, and a member variable name.

Returns:

(Varies)

Description:

Retrieves the value of a Java object's data member. The first argument is the object (or the name of a class, for a static member) and the second argument is the name of the field. The return value is converted to a suitable Jess value exactly as for call.

16.81. (get-multithreaded-io)

Arguments:

None

Returns:

Boolean

Description:

Returns TRUE is Jess is currently using a separate thread to flush I/O streams. Turning this on can lead to a modest performance enhancement, at the expense of possible loss of output on program termination.

16.82. (get-reset-globals)

Arguments:

None

Returns:

Boolean

Description:

Indicates the current setting of global variable reset behavior. See set-reset-globals for an explanation of this property.

16.83. (get-salience-evaluation)

Arguments:
None
Returns:
Symbol
Description:
Indicates the current setting of salience evaluation behavior. See set-salience-evaluation for an explanation of this property.

16.84. (get-strategy)

Arguments:
(None)
Returns:
A symbol, the name of the current conflict resolution strategy.
Description:
Returns the name of the current conflict resolution strategy. See `set-strategy`.

16.85. (halt)

Arguments:
None
Returns:
TRUE
Description:
Halts rule execution. No effect unless called from the RHS of a rule.

16.86. (help <function-name>)

Arguments:
A function name
Returns:
nil
Description:
Prints a description of the named function to WSTDOUT.

16.87. (if <expression> then <action>* [elif <expression> then <action>*]* [else <action>*])

Arguments:
A Boolean expression, the symbol `then`, and any number of additional expressions; followed (zero or more times) by the symbol `elif`, a Boolean expression, `then`, and another list of expressions; optionally followed by the symbol `else` and another list of expressions.
Returns:
(Varies)
Description:
Allows conditional execution of a group of actions. The first Boolean expression is evaluated. If it does not evaluate to `FALSE`, the first list of actions is evaluated, and the return value is that returned by the last action of that list. If it does evaluate to `FALSE`, and there are optional `elif`

blocks, then the Boolean expression of each of these is evaluated in turn; the first time one of them evaluates to non-FALSE, the associated list of actions is evaluated and the last result is returned. Finally, if none of the expressions is non-FALSE, and the optional `else` block is supplied, then the final list of actions is evaluated and the value of the last is returned.
Example:

```
Jess> (if (> ?x 100) then
    (printout t "X is big" crlf)
 elif (> ?x 50) then
    (printout t "X is average" crlf)
 else
    (printout t "X is small" crlf))
```

16.88. (implement <interface> [using] <function>)

Arguments:
The name of an interface, the optional symbol "using", and a Jess function or lambda expression
Returns:
An object that implements the given interface
Description:
Lets you implement any interface from Jess. Here's an example of creating a Runnable object entirely from Jess and running it in a new Thread:

```
Jess> ;; Function's arguments will be the name of the method called on proxy object
;; (run, here ), followed by the individual arguments passed to called
;; method (none here).
(deffunction my-runnable ($?args)
  (printout t "Hello, World" crlf))

TRUE

Jess> ;; Make a Runnable whose run() method will call my-runnable
(bind ?runnable (implement Runnable using my-runnable))

Jess> ;; Use the Runnable
((new Thread ?runnable) start)
```

16.89. (implode$ <list-expression>)

Arguments:
One list
Returns:
String
Description:
Creates a string from a list value. Converts each element of the list to a string, and returns these strings concatenated with single intervening spaces.

16.90. (import <symbol>)

Arguments:
One symbol
Returns:
TRUE
Description:

Works like the Java `import` statement. You can import either a whole package using

```
Jess> (import java.io.*)
```

or a single class using

```
Jess> (import java.awt.Button)
```

After that, all functions that can accept a Java class name (new, defclass, call, etc) will refer to the import list to try to find the class that goes with a specific name. Note that java.lang.* is now implicitly imported.

In addition, when you import a single class by name, Jess will define a series of Userfunctions that provide easy access to that class's static members. These functions are named *ClassName.memberName*. For example, because the classes in java.lang are imported this way, there are functions named "Thread.currentThread", "Integer.parseInt" which give access to those Java methods; there are also functions named "Short.MAX_VALUE" and "Thread.NORM_PRIORITY" which return the values of those Java constants.

```
Jess> (if (> ?i (Short.MAX_VALUE)) then
    ((System.out) println "Too large for short")
    else
    ((System.out) println "Would fit in a short"))
```

16.91. (insert$ <list-expression> <integer-expression> <single-or-list-expression>+)

Arguments:
A list, an integer, and one or more lists
Returns:
A list
Description:
Returns a new list like the original but with one or more values inserted. Inserts the elements of the second and later lists so that they appear starting at the given 1-based index of the first list. An index value one greater than the list length is legal and will result in the second list being appended at the end of the first list.

16.92. (instanceof <Java object> <class-name>)

Arguments:
A Java object and the name of a Java class
Returns:
Boolean
Description:
Returns true if the Java object can be assigned to a variable whose type is given by the class name. Implemented using java.lang.Class.isInstance() . The class name can be fully-qualified or it can be an imported name; see the discussion of the import function.

16.93. (integer <numeric-expression>)

Arguments:
One numeric expression
Returns:
Integer
Description:
Converts its only argument to an integer. Truncates any fractional component of the value of the given numeric expression and returns the integral part.

16.94. (integerp <expression>)

Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE for integers; otherwise, returns FALSE.

16.95. (intersection$ <list-expression> <list-expression>)

Arguments:
Two lists
Returns:
List
Description:
Returns the intersection of two lists. Returns a list consisting of the elements the two argument lists have in common.

16.96. (java-objectp <expression>)

Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE if the expression evaluates to a Java object.

16.97. (jess-type <value>)

Arguments:
Any value
Returns:
Symbol
Description:
Returns a symbol denoting the Jess data type of the argument.

16.98. (jess-version-number)

Arguments:
None
Returns:
Float
Description:
Returns a version number for Jess; currently 7.0 .

16.99. (jess-version-string)

Arguments:
None
Returns:
String
Description:

Returns a human-readable string descriptive of this version of Jess.

16.100. (lambda (<arguments>) <function call>+)

Arguments:
A list of arguments followed by any number of function calls
Returns:
An anonymous deffunction
Description:
Lets you create an unnamed deffunction. This is useful in conjunction with the `implement` function. In this example, we create a Runnable and execute it in a Thread.

```
Jess> ((new Thread (implement Runnable using
     (lambda ($?args)
         (printout t "Hello, World" crlf))
     )
  ) start)
```

This looks a lot like doing the same thing with an anonymous class in Java:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello, World!");
    }
}.start();
```

16.101. (length$ <list-expression>)

Arguments:
List
Returns:
Integer
Description:
Returns the number of elements in a list value.

16.102. (lexemep <expression>)

Arguments:
Any expression
Returns:
Boolean
Description:
Returns TRUE for symbols and strings; otherwise, returns FALSE.

16.103. (list <value>*)

Arguments:
Any number of values
Returns:
A list
Description:
An alias for `create$`.

16.104. (list-deftemplates [module-name | *])

Arguments:
Optionally, a module name, or the symbol "*"
Returns:
nil
Description:
With no arguments, prints a list of all `deftemplate`s in the current module (not the focus module) to the 't' router. With a module name for an argument, prints the names of the templates in that module. With "*" as an argument, prints the names of all templates.

16.105. (list-focus-stack)

Arguments:
None
Returns:
nil
Description:
Displays the module focus stack, one module per line; the top of the stack (the focus module) is displayed first.

16.106. (list-function$)

Arguments:
None
Returns:
List
Description:
Returns a list list of all the functions currently callable, including intrinsics, deffunctions, and `jess.Userfunction`s. Each function name is a symbol. The names are sorted in alphabetical order.

16.107. (listp <expression>)

Arguments:
Any value
Returns:
Boolean
Description:
Returns `TRUE` for list values; otherwise, returns `FALSE`.

16.108. (load-facts <file-name>)

Arguments:
A string representing the name of a file of facts
Returns:
Boolean
Description:

Asserts facts loaded from a file. The argument should name a file containing a list of facts (not `deffacts` constructs, and no other commands or constructs). Jess will parse the file and assert each fact. The return value is the return value of assert when asserting the last fact. In an applet, `load-facts` will use `getDocumentBase()` to find the named file.

Note: See the `batch` command for a discussion about specifying filenames in Jess.

The file can be in either of two formats. The first format is just a list of facts in Jess language syntax. The second format is an XML file with a `fact-list` root element containing nothing but `fact` elements, as in JessML.

16.109. (load-function <class-name>)

Arguments:
The name of a Java class
Returns:
Boolean
Description:
The argument must be the fully-qualified name of a Java class that implements the `jess.Userfunction` interface. The class is loaded in to Jess and added to the engine, thus making the corresponding command available. See Extending Jess with Java for more information.

16.110. (load-package <class-name>)

Arguments:
The name of a Java class
Returns:
Boolean
Description:
The argument must be the fully-qualified name of a Java class that implements the `jess.Userpackage` interface. The class is loaded in to Jess and added to the engine, thus making the corresponding package of commands available. See Extending Jess with Java for more information.

16.111. (log <numeric-expression>)

Arguments:
One numeric expression
Returns:
Number
Description:
Returns the logarithm base `e` of its only argument.

16.112. (log10 <numeric-expression>)

Arguments:
One numeric expression
Returns:
Number
Description:
Returns the logarithm base-10 of its only argument.

16.113. (long <expression>)

Arguments:
One expression, either numeric or String
Returns:
RU.LONG

Description:
Interprets the expression as a Java long (if possible) and returns a long value. This function is retained for backward compatibility only, as the Jess language now allows long literals.

16.114. (longp <expression>)

Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE if the expression is of type RU.LONG; FALSE otherwise.

16.115. (lowcase <lexeme-expression>)

Arguments:
One symbol or string.
Returns:
String or symbol
Description:
Converts uppercase characters in a string or symbol to lowercase. Returns the argument as an all-lowercase symbol unless the argument is a string, in which case a string is returned.

16.116. (map <function> <list>)

Arguments:
A function and a list
Returns:
A list
Description:
Calls the function on each item in the list; returns a list of all the results. The function can either be the name of a Userfunction, or it can be a `lambda` expression.

16.117. (matches <lexeme-expression>)

Arguments:
One symbol, a rule or query name
Returns:
TRUE
Description:
Produces a printout, useful for debugging, of the contents of the left and right Rete memories of each two-input node on the given rule or query's LHS.

16.118. (max <numeric-expression>+)

Arguments:
One or more numerical expressions
Returns:
Number
Description:
Returns the value of its largest numeric argument

16.119. (member$ <expression> <list-expression>)

Arguments:
A value and a list
Returns:
Integer or FALSE
Description:
Returns the first position (1-based index) of a value within a list; otherwise, returns FALSE.

16.120. (min <numeric-expression>+)

Arguments:
One or more numeric expressions
Returns:
Number
Description:
Returns the value of its smallest numeric argument.

16.121. (mod <numeric-expression> <numeric-expression>)

Arguments:
Two integer expressions
Returns:
Integer
Description:
Returns the remainder of the result of dividing the first argument by its second (assuming that the result of the division must be an integer).

16.122. (modify <fact-specifier> (<slot-name> <value>)+)

Arguments:
A Fact and one or more two-element lists
Returns:
A Fact
Description:

Modifies a given unordered fact in working memory. The first argument specifies the fact to modify, and can be either a Fact object (such as you'd obtain from a pattern binding) or an integer (the id number of a fact.) If you use the id number, the corresponding fact must be looked up using `jess.Rete.findFactByID(int)`, a slow operation. The fact must be an unordered fact.

Subsequent arguments are two-item lists. Each list is taken as the name of a slot in this fact and a new value to assign to the slot. The fact is removed from working memory, the values in the specified slots are replaced with the new values, and the fact is reasserted. The fact-ID of the fact does not change. The fact itself is returned. A `jess.JessEvent` of type FACT + MODIFIED will be sent if the event mask is set appropriately.

Modifying a shadow fact will cause the appropriate object properties to be set as well.

As of Jess version 7, the slot names can be variables.

It is important to remember that all pattern-matching happens during calls to `assert`, `retract`, `modify`, and related functions. Pattern-matching happens whether the engine is running or not.

16.123. (multifieldp <expression>)

Arguments:
Any value
Returns:
Boolean
Description:
Deprecated. Use `listp` instead.

16.124. (neq <expression> <expression>+)

Arguments:
Two or more values
Returns:
Boolean
Description:

Returns TRUE if the first argument is not equal in type and value to all subsequent arguments (see `eq`).

While often used in procedural code, this function is only rarely used during pattern matching. Direct matching is preferable.

16.125. (new <class-name> <argument>*)

Arguments:
The name of a Java class and zero or more expressions
Returns:
Boolean
Description:
Creates a new Java object and returns it. The first argument is the class name: `java.util.Vector`, for example. The second and later arguments are constructor arguments. The constructor will be chosen from among all constuctors for the named class based on a *first-best fit* algorithm. Built-in Jess types are converted as necessary to match available constructors. See the text for more details. Also see the `import` function.

16.126. (not <expression>)

Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE if its only arguments evaluates to FALSE; otherwise, returns FALSE. Note that this function is distinct from the not conditional element.

16.127. (nth$ <integer-expression> <list-expression>)

Arguments:
A number and a list
Returns:
(Varies)
Description:

Returns the value of the specified (1-based index) field of a list value.

16.128. (numberp <expression>)

Arguments:
One expression
Returns:
Boolean
Description:
Returns TRUE for numbers; otherwise, returns FALSE.

16.129. (oddp <integer-expression>)

Arguments:
One integer expression
Returns:
Boolean
Description:
Returns TRUE for odd numbers; otherwise, returns FALSE; see evenp.

16.130. (open <file-name> <router-identifier> [r|w|a])

Arguments:
A file name, an identifier for the file (a symbol), and optionally a mode string, one of r, w, a.
Returns:
The file identifier, a router name.
Description:
Opens a file. Subsequently, the given router identifier can be passed to printout, read, readline, or any other functions that accept I/O routers as arguments. By default, the file is opened for reading; if a mode string is given, it may be opened for reading only (r), writing only (w), or appending (a).
Note: See the batch command for a discussion about specifying filenames in Jess.

16.131. (or <expression>+)

Arguments:
One or more expressions
Returns:
Boolean
Description:
Returns TRUE if any of the arguments evaluates to a non-FALSE value; otherwise, returns FALSE. Note that this function is distinct from the or conditional element.

16.132. (pi)

Arguments:
None
Returns:
Number
Description:
Returns the number pi.

16.133. (pop-focus)

Arguments:
None
Returns:
The name of a module
Description:
Removes the top module from the focus stack and returns its name.

16.134. (ppdeffacts <symbol>)

Arguments:
The name of a deffacts
Returns:
String
Description:
Returns a pretty-print rendering of a `deffacts`.

16.135. (ppdeffunction <symbol>)

Arguments:
The name of a deffunction
Returns:
String
Description:
Returns a pretty-print representation of a `deffunction`.

16.136. (ppdefglobal <symbol>)

Arguments:
The name of a defglobal
Returns:
String
Description:
Returns a pretty-print representation of a `defglobal`

16.137. (ppdefquery <symbol> | *)

Arguments:
The name of a defquery or *
Returns:
String
Description:
An alias for `ppdefrule` .

16.138. (ppdefrule <symbol> | *)

Arguments:
The name of a rule or query, or the symbol *
Returns:
String
Description:

Returns a pretty-print rendering of a `defrule` or `defquery`. If the argument is the symbol "*", a single string containing all existing rules and queries in alphabetical order of their names is returned.

16.139. (ppdeftemplate <symbol>)

Arguments:
The name of a template
Returns:
String
Description:
Returns a pretty-print representation of a `deftemplate`.

16.140. (printout <router-identifier> <expression>*)

Arguments:
A router identifier followed by zero or more expressions
Returns:
nil
Description:
Sends unformatted output to the specified logical name. Prints its arguments to the named router, which must be open for output. No spaces are added between arguments. The special symbol `crlf` prints as a newline. The special router name `t` can be used to signify standard output.

16.141. (progn <expression>*)

Arguments:
Zero or more expressions
Returns:
The result of evaluating the last expression, or nil
Description:
A simple control structure that allows you to group multiple function calls where syntactically only one is allowed - for instance, on the LHS of a rule.

16.142. (provide <symbol>)

Arguments:
A symbol
Returns:
Returns the symbol.
Description:

Provides a feature to Jess. The symbol is entered in the feature table. See `require`.

If the feature is meant to be loaded from a subdirectory of the current directory or a subdirectory of a class path root, then the feature name must reflect that. For example if a feature X is implemented in the file `com/company/X.clp`, then the feature name must be "com/company/X". This full feature name must be used by `require` to load the feature or the feature may not be found.

16.143. (random)

Arguments:
None
Returns:
Number
Description:
Returns a pseudo-random integer between 0 and 65536.

16.144. (read [<router-identifier>])

Arguments:
An optional input router identifier (when omitted t is the default)
Returns:
(Varies)
Description:
Reads a single-field value from a specified logical name. Read a single symbol, string, or number from the named router, returns this value. The router `t` means standard input. Newlines are treated as ordinary whitespace. If you need to parse text line-by-line, use `readline` and `explode$`.

16.145. (readline [<router-identifier>])

Arguments:
An optional input router identifier (when omitted t is the default)
Returns:
String
Description:
Reads an entire line as a string from the specified logical name (router). The router `t` means standard input.

16.146. (regexp <regular expression> <data>)

Arguments:
A regular expression and a target, as symbols or strings
Returns:
Boolean
Description:
Compiles the regular expression and tries to match it against the entire target string, returning the Boolean result of the match. Uses the `java.util.regexp` package.

16.147. (remove <symbol>)

Arguments:
A symbol, the name of a template
Returns:
(Nothing)
Description:

Retracts all the facts currently in working memory that use the given template. Removing a shadow fact template will result in an implicit call to `undefinstance` for the corresponding objects (the objects will no longer be pattern-matched). A `jess.JessEvent` of type FACT + REMOVED will be sent for each fact removed if the event mask is set appropriately.

It is important to remember that all pattern-matching happens during calls to assert, retract, modify, and related functions. Pattern-matching happens whether the engine is running or not.

16.148. (replace$ <list-expression> <begin-integer-expression> <end-integer-expression> <expression>+)

Arguments:
A list, two numeric expressions, and one or more additional single or list values
Returns:
List
Description:
Returns a copy of a the original list with the elements in a specified range replaced with a new set of values. The variable number of final arguments are inserted into the first list, replacing elements between the 1-based indices given by the two numeric arguments, inclusive. Example:

```
Jess> (replace$ (create$ a b c) 2 2 (create$ x y z))

(a x y z c)
```

16.149. (require <symbol> [<filename>])

Arguments:
A symbol, and optionally a filename
Returns:
Returns the symbol, or throws an exception on failure.
Description:

"Require" is similar to batch. The main difference is that it will only load a file once. If a file has been read before, this function won't read it a second time, whereas batch would.

The symbol argument is a "feature name". A file can "provide a feature" (see the provide function.) Once a feature has been provided, subsequent calls to `require` for this same feature name will be ignored.

If the optional filename is supplied, Jess passes that name to batch to attempt to read the file, if necessary. If the optional filename is not provided, Jess appends ".clp" to the feature name and uses that as the argument to batch. If the feature is not provided by the first file that is read, an exception is thrown to announce this failure.

The most important use of "require" is to establish dependencies between files in the JessDE editor.

16.150. (require* <symbol> [<filename>])

Arguments:
A symbol, and optionally a filename
Returns:
Returns the symbol, or nil on failure.
Description:

This function is just like require, except that it fails silently if the file is missing or if there is an error while parsing the required file. If the feature is provided, the feature name is returned; otherwise, `nil` is returned instead.

16.151. (reset)

Arguments:
None
Returns:
TRUE
Description:
Removes all facts from working memory, removes all activations, then asserts the fact `(initial-fact)`, then asserts all facts found in deffacts, asserts a fact representing each registered Java object, and (if the set-reset-globals property is TRUE) initializes all defglobals.

16.152. (rest$ <list-expression>)

Arguments:
One list
Returns:
List
Description:
Returns all but the first field of a list as a new list.

16.153. (retract <expression>+)

Arguments:
One or more `jess.Fact` objects or integers
Returns:
TRUE
Description:

Retracts the facts given. Retracting a shadow fact will result in an implicit call to `undefinstance` for the corresponding object (the object will no longer be pattern-matched). A `jess.JessEvent` of type FACT + REMOVED will be sent if the event mask is set appropriately. Note that the arguments to this function must be integers or actual `jess.Fact` objects; they cannot be explicit facts as are accepted by `assert`.

It is important to remember that all pattern-matching happens during calls to `assert`, `retract`, `modify`, and related functions. Pattern-matching happens whether the engine is running or not.

16.154. (retract-string <string>)

Arguments:
A string, a representation of a Fact
Returns:
TRUE
Description:
Parses the string as a Fact; if such a fact exists in working memory, calls `retract` on it.

16.155. (return [<expression>])

Arguments:
An optional expression
Returns:
(Varies)
Description:

From a deffunction, returns the given value and exits the deffunction immediately. From the RHS of a defrule, terminates the rule's execution immediately and pops the current focus module from the focus stack. No argument should be given when return is called from the RHS of a rule.

16.156. (round <numeric-expression>)

Arguments:
One numeric expression
Returns:
Integer
Description:
Rounds its argument to the closest integer.

16.157. (rules [<module-name> | *])

Arguments:
Optionally, a module name, or the symbol "*"
Returns:
nil
Description:
With no arguments, prints a list of all rules and queries in the current module (not the focus module) to the 't' router. With a module name for an argument, prints the names of the rules and queries in that module. With "*" as an argument, prints the names of all rules and queries.

16.158. (run [<integer>])

Arguments:
Optionally, a single integer
Returns:
Integer
Description:
Starts the inference engine. If no argument is supplied, Jess will keep running until no more activations remain or halt is called. If an argument is supplied, it gives the maximum number of rules to fire before stopping. The function returns the number of rules actually fired.

16.159. (run-query <query-name> <expression>*)

Arguments:
The name of a query, and zero or more additional expressions
Returns:
A java.util.Iterator
Description:

Deprecated. Use run-query* instead.

Runs a query and returns a java.util.Iterator of the matches. See the documentation for defquery for more details. Note that run-query can lead to backwards chaining, which can cause rules to fire; thus if run-query is called on a rule RHS, other rules' RHSs may run to completion before the instigating rule completes. Putting run-query on a rule RHS can also cause the count of executed rules returned by run to be low.

Note that the Iterator returned by this function should be used immediately. It will become invalid if any of the following functions are called before you've used it: reset, count-query-results, or

`run-query`. It *may* become invalid if any of the following are called: `assert`, `retract`, `modify`, or `duplicate`, and if any of the affected facts are involved in the active query's result.

16.160. (run-query* <query-name> <expression>*)

Arguments:

The name of a query, and zero or more additional expressions

Returns:

A `jess.QueryResult`

Description:

Runs a query and returns a `jess.QueryResult` of the matches. See the documentation for defquery for more details. Note that run-query can lead to backwards chaining, which can cause rules to fire; thus if run-query is called on a rule RHS, other rules' RHSs may run to completion before the instigating rule completes. Putting `run-query*` on a rule RHS can also cause the count of executed rules returned by `run` to be low.

16.161. (run-until-halt)

Arguments:

None.

Returns:

Integer

Description:

Runs the engine until `halt` is called. Returns the number of rules fired. When there are no active rules, the calling thread will be blocked waiting on the activation semaphore.

16.162. (save-facts <file-name> [<template-name>])

Arguments:

A filename, and optionally a symbol

Returns:

Boolean

Description:

Saves facts to a file in Jess language format. Attempts to open the named file for writing, and then writes a list of all facts in working memory to the file. This file is suitable for reading with `load-facts`. If the optional second argument is given, only facts whose head matches this symbol will be saved.

Note: See the `batch` command for a discussion about specifying filenames in Jess.

16.163. (save-facts-xml <file-name> [<template-name>])

Arguments:

A filename, and optionally a symbol

Returns:

Boolean

Description:

Saves facts to a file in XML format. This function attempts to open the named file for writing, and then writes a list of all facts in working memory to the file. A well-formed document containing an XML declaration, a `fact-list` root element, and one `fact` element for each fact will be written.

This file is suitable for reading with load-facts. If the optional second argument is given, only facts whose head matches this symbol will be saved.

Note: See the batch command for a discussion about specifying filenames in Jess.

16.164. (set <Java object> <string-expression> <expression>)

Arguments:
A Java object, a property name, and an expression
Returns:
The last argument
Description:
Sets a JavaBean's property or instance variable to the given value. The first argument is the Bean object; the second argument is the name of the property or variable. The third value is the new value for the property; the same conversions are applied as for new and call. Jess will first try to find a JavaBean property by the given name; if none is found, it will look for an instance variable.

16.165. (set-current-module <module-name>)

Arguments:
The name of a valid module
Returns:
The name of the previous current module
Description:
Sets the current module. Any constructs defined without explicitly naming a module are defined in the current module. Note that defining a defmodule also sets the current module.

16.166. (set-factory <factory object>)

Arguments:
An object that implements the interface jess.factory.Factory
Returns:
A jess.factory.Factory
Description:
Set the "thing factory" for the active Rete object. Providing an alternate "thing factory" is a very advanced, and currently undocumented, way to extend Jess's functionality.

16.167. (setgen <numeric-expression>)

Arguments:
A numeric expression
Returns:
TRUE
Description:
Sets the starting number used by gensym*. Note that if this number has already been used, gensym* uses the next larger number that has not been used.

16.168. (set-member (<Java object> | <string-expression>) <string> <expression>)

Arguments:
A Java object or class name, a member variable name and an expression
Returns:

The last argument

Description:

Sets a Java object's member variable to the given value. The first argument is the object (or the name of the class, in the case of a static member variable). The second argument is the name of the variable. The third value is the new value for the variable; the same conversions are applied as for `new` and `call`.

16.169. (set-multithreaded-io (TRUE | FALSE))

Arguments:

Boolean

Returns:

Boolean

Description:

Specify whether Jess should use a separate thread to flush I/O streams. Turning this on can lead to a modest performance enhancement, at the expense of possible loss of output on program termination. Returns the previous value of this property.

16.170. (set-node-index-hash <integer>)

Arguments:

One integral value

Returns:

TRUE

Description:

Sets the default hashing key used in all Rete network join node memories defined after the function is called; this function will not affect parts of the network already in existence at the time of the call. A small value will give rise to memory-efficient nodes; a larger value will use more memory. If the created nodes will generally have to remember many partial matches, large numbers will lead to faster performance; the opposite may be true for nodes which will rarely hold more than one or two partial matches. This function sets the default; explicit `declare` statements can override this for specific rules.

16.171. (set-reset-globals <Boolean>)

Arguments:

One Boolean value

Returns:

Boolean

Description:

Changes the current setting of the global variable reset behavior. If this property is set to TRUE (the default), then the (reset) command reinitializes the values of global variables to their initial values (if the initial value was a function call, the function call is reexecuted.) If the property is set to FALSE, then (reset) will not affect global variables. Note that in previous versions of Jess, defglobals were always reset; but if the initial value was set with a function call, the function was **not** reevaluated. Now it is.

16.172. (set-salience-evaluation (when-defined | when-activated | every-cycle))

Arguments:

One of the symbols `when-defined`, `when-activated`, or `every-cycle`

Returns:

One of the potential arguments (the previous value of this property)

Description:

Changes the current setting of the salience evaluation behavior. By default, a rule's salience will be determined once, when the rule is defined (when-defined.) If this property is set to when-activated, then the salience of each rule will be redetermined immediately before each time it is placed on the agenda. If the property is set to every-cycle, then the salience of every rule is redetermined immediately after each time any rule fires.

16.173. (set-strategy <strategy-name>)

Arguments:

A symbol or string representing the name of a strategy (can be a fully-qualifed Java class name). You can use `depth` and `breadth` to represent the two built-in strategies.

Returns:

The previous strategy as a symbol.

Description:

Lets you specify the *conflict resolution strategy* Jess uses to order the firing of rules of equal salience. Currently, there are two strategies available: *depth (LIFO)* and *breadth (FIFO).* When the depth strategy is in effect (the default), more recently activated rules are fired before less recently activated rules of the same salience. When the breadth strategy is active, rules of the same salience fire in the order in which they are activated. Note that in either case, if several rules are activated simultaneously (i.e., by the same fact-assertion event) the order in which these several rules fire is unspecified, implementation-dependent and subject to change. More built-in strategies may be added in the future. You can implement your own strategies in Java by creating a class that implements the `jess.Strategy` interface and then specifying its fully-qualified classname as the argument to `set-strategy`. Details can be gleaned from the source.

16.174. (set-value-class <string-expression> TRUE|FALSE)

Arguments:

The name of a Java class

Returns:

TRUE

Description:

A *value object* is an instance of a class that represents a specific value. They are often immutable like Integer, Double, and String. For Jess's purposes, a value object is one whose `hashCode()` method returns a constant -- i.e., whose hash code doesn't change during normal operation of the class. Integer, Double, and all the other wrapper classes qualify, as does String, and generally all immutable classes. Any class that doesn't override the default `hashCode()` method also qualifies as a value object by the definition. Java's Collection classes (Lists, Maps, Sets, etc.) are classic examples of classes that are *not* value objects, because their hash codes depend on the collection's contents.

As far as Jess is concerned, an object is a value object as long as its hash code won't change while the object is in working memory. This includes the case where the object is contained in a slot of any fact. If the hash code will only change during calls to `modify`, then the object is still a value object.

Jess can make certain assumptions about value objects that lead to large performance increases during pattern matching. Because many classes are actually value classes by Jess's broad definition, *Jess now assumes that all objects (except for Maps and Collections) are value objects by default.* If you're working with a class that is *not* a value class, it's very important that you tell Jess about it by using the set-value-class function or the static method `jess.HashCodeComputer.setIsValueClass(Rete, String, boolean)` Failure to do so will lead to undefined (bad) behavior.

16.175. (set-watch-router <router-name>)

Arguments:
A symbol, the name of a valid output router
Returns:
The previous watch router name
Description:
Sets the router that the output from `watch` goes to. The old value is returned. Note that the watch router is not reset by `reset` or `clear`.

16.176. (show-deffacts)

Arguments:
None
Returns:
nil
Description:
Displays all defined `deffacts` to the 't' router.

16.177. (show-deftemplates)

Arguments:
None
Returns:
nil
Description:
Displays all defined `deftemplate`s to the 't' router.

16.178. (show-jess-listeners)

Arguments:
None
Returns:
nil
Description:
Displays all `jess.JessListener`s registered with the engine to the 't' router.

16.179. (socket <Internet-hostname> <TCP-port-number> <router-identifier>)

Arguments:
An Internet hostname, a TCP port number, and a router identifier
Returns:
The router identifier
Description:
Somewhat equivalent to `open`, except that instead of opening a file, opens an unbuffered TCP network connection to the named host at the named port, and installs it as a pair of read and write routers under the given name.

16.180. (sqrt <numeric-expression>)

Arguments:
A numeric expression
Returns:

Number

Description:

Returns the square root of its only argument.

16.181. (store <string or symbol> <expression>)

Arguments:

A string or symbol and any other value

Returns:

(varies)

Description:

Associates the expression with the name given by the first argument, such that later calls to the `fetch` will retrieve it. Storing the symbol nil will clear any value associated with name. Analagous to the `store()` member function of the `jess.Rete` class. See section on using store and fetch for more details.

16.182. (str-cat <expression>*)

Arguments:

Zero or more expressions

Returns:

String

Description:

Concatenates its arguments as strings to form a single string. For Java objects, the toString() method of the contained object is called.

16.183. (str-compare <string-expression> <string-expression>)

Arguments:

Two symbols or strings

Returns:

Integer

Description:

Lexicographically compares two strings. Returns 0 if the strings are identical, a negative integer if the first is lexicographically less than the second, a positive integer if lexicographically greater.

16.184. (str-index <lexeme-expression> <lexeme-expression>)

Arguments:

Two symbols or strings

Returns:

Integer or FALSE

Description:

Returns the position of the first argument within the second argument. This is the 1-based index at which the first string first appears in the second; otherwise, returns FALSE.

16.185. (stringp <expression>)

Arguments:

One expression

Returns:

Boolean

Description:

Returns TRUE for strings; otherwise, returns FALSE.

16.186. (str-length <lexeme-expression>)

Arguments:
A symbol or string
Returns:
Integer
Description:
Returns the length of a symbol in characters.

16.187. (subseq$ <list-expression> <begin-integer-expression> <end-integer-expression>)

Arguments:
A list and two numeric expressions
Returns:
List
Description:
Extracts the specified range from a list value consisting of the elements between the two 1-based indices of the given list, inclusive. Index values less than 1 and greater than the list length are accepted, and so is a begin index greater than an end index, resulting in an empty list.

16.188. (subsetp <list-expression> <list-expression>)

Arguments:
Two lists
Returns:
Boolean
Description:
Returns TRUE if the first argument is a subset of the second (i.e., all the elements of the first list appear in the second list); otherwise, returns FALSE.

16.189. (sub-string <begin-integer-expression> <end-integer-expression> <string-expression>)

Arguments:
Two numbers and a string
Returns:
String
Description:
Retrieves a subportion from a string. Returns the string consisting of the characters between the two 1-based indices of the given string, inclusive. Both index values must not be less than 1 or greater than the string length. A begin index that is equal to the end index plus one results in the null string.

16.190. (symbolp <expression>)

Arguments:
One expression
Returns:
Boolean
Description:

Returns TRUE for symbols; otherwise, returns FALSE.

16.191. (sym-cat <expression>*)

Arguments:
Zero or more expressions
Returns:
Symbol
Description:
Concatenates its arguments as strings to form a single symbol. For Java objects, the toString() method of the contained object is called.

16.192. (synchronized <java-object> <action>*)

Arguments:
Any Java object, followed by any number of expressions
Returns:
(varies)
Description:
Executes the expressions inside a Java "synchronized" block which locks the given object. Returns the value of the last expression evaluated.

16.193. (system <lexeme-expression>+ [&])

Arguments:
One or more symbols or strings
Returns:
a `java.lang.Process` object, or FALSE
Description:
Sends a command to the operating system. Each symbol or string becomes one element of the argument array in a call to the Java `java.lang.Runtime.exec(String[] cmdaray)` method; therefore to execute the command `edit myfile.txt`, you should call `(system edit myfile.txt)`, not (system "edit myfile.txt").
Normally blocks (i.e., Jess stops until the launched application returns), but if the last argument is an ampersand (`&`), the program will run in the background. The standard output and standard error streams of the process are connected to the 't' router, but the input of the process is not connected to the terminal.
Returns the Java Process object. You can call `waitFor` and then `exitValue` to get the exit status of the process.

16.194. (throw <java-object>)

Arguments:
A Java object that must inherit from `java.lang.Throwable`
Returns:
Does not return
Description:
Throws the given exception object. If the object is a `jess.JessException`, throws it directly. If the object is some other type of exception, it is wrapped in a `JessException` before throwing. The object's stack trace is filled in such that the exception will appear to have been created by the `throw` function.

16.195. (time)

Arguments:
None
Returns:
Number
Description:
Returns the number of seconds since 12:00 AM, Jan 1, 1970.

16.196. (try <expression>* [catch <expression>*] [finally <expression>*])

Arguments:
One or more expressions, followed optionally by the symbol `catch` followed by zero or more expressions, followed optionally by the symbol `finally` followed by zero or more expressions. Either the `catch`, or the `finally`, or both must be included.
Returns:
(Varies)
Description:
This command works something like Java `try` with a few simplifications. The biggest difference is that the `catch` clause can specify neither a type of exception nor a variable to receive the exception object. All exceptions occurring in a `try` block are routed to the single `catch` block. The variable `?ERROR` is made to point to the exception object. For example:

```
Jess>
(try
    (open NoSuchFile.txt r)
 catch
    (printout t (call ?ERROR toString) crlf))
```

prints

```
Jess reported an error in routine open
    while executing (open NoSuchFile.txt r).
Message: I/O Exception.
```

An empty `catch` block is fine. It just signifies ignoring possible errors.
The code in the `finally` block, if present, is executed after all try and/or catch code has executed, immediately before the `try` function returns.

16.197. (undefadvice <function-name> | ALL | <list>)

Arguments:
A function name, or ALL, or a list of function names
Returns:
TRUE
Description:
Removes all advice from the named function(s).

16.198. (undeffacts <deffacts-name> | *)

Arguments:
The name of a deffacts, or the symbol "*"
Returns:
Boolean
Description:
Deletes a deffacts. The next time the engine is reset, the facts in that deffacts will not be asserted. If the argument is "*", all deffacts are deleted.

16.199. (undefinstance (<java-object> | *))

Arguments:
A Java object, or the symbol "*"
Returns:
TRUE
Description:
If the object currently has a shadow fact, it is removed from the working memory. Furthermore, if the object has a `java.beans.PropertyChangeListener` installed, this is removed as well. If the argument is "*" this is done for all Java objects in working memory.

16.200. (undefrule <rule-name>)

Arguments:
The name of a rule
Returns:
Boolean
Description:
Deletes a rule. Removes the named rule from the Rete network and returns `TRUE` if the rule existed. This rule will never fire again.

16.201. (union$ <list-expression>+)

Arguments:
One or more lists
Returns:
List
Description:
Returns a new list consisting of the union of all of its list arguments (i.e., of all the elements that appear in any of the arguments with duplicates removed).

16.202. (unwatch <symbol>)

Arguments:
One or more of the symbols `all`, `rules`, `compilations`, `activations`, `facts`, `focus`
Returns:
Description:
Causes trace output to not be printed for the given indicators. See `watch`.

16.203. (upcase <lexeme-expression>)

Arguments:
A string or symbol
Returns:
A string or symbol
Description:
Converts lowercase characters in a string or symbol to uppercase. Returns the argument as an all-uppercase symbol, unless the argument is a string, in which case a string is returned.

16.204. (update <java-object>+)

Arguments:
One or more Java objects, previously passed as arguments to `add` or `definstance`.
Returns:
The shadow fact tied to the last argument.
Description:
Java objects in working memory aren't necessarily updated automatically, since Jess may not know when an object object has been changed. This function lets you tell Jess explicitly that one or more Java objects have been updated. In response, Jess will find their corresponding shadow facts and update all their slots.

16.205. (view)

Arguments:
None
Returns:
TRUE
Description:
The view command displays a live snapshot of the Rete network in a graphical window. See How Jess Works for details.

16.206. (watch <symbol>)

Arguments:
One or more of the symbols `all`, `rules`, `compilations`, `activations`, `facts`, `focus`
Returns:
TRUE
Description:
Produces additional debug output when specific events happen in Jess, depending on the argument(s). Any number of different watches can be active simultaneously:

- `rules`: prints a message when any rule fires.
- `compilations`: prints a message when any rule is compiled.
- `activations`: prints a message when any rule is activated, or deactivated, showing which facts have caused the event.
- `facts`: print a message whenever a fact is asserted or retracted.
- `focus`: print a message for each change to the module focus stack.
- `all`: all of the above.

16.207. (while <expression> [do] <action>*)

Arguments:
A Boolean expression, the symbol `do`, and zero or more expressions
Returns:
(Varies)
Description:
Allows conditional looping. Evaluates the boolean expression repeatedly. As long as it does not equal `FALSE`, the list of other expressions are evaluated. A `break` also terminates the iteration. The value of the last expression evaluated is the return value of this function.

17. Jess Constructs

A *construct* is something that looks like a function, but isn't one. It generally has odd syntax that can't be used in a regular function call. Most of the names starting with "def-" in Jess are construct names: defrule, defglobal, etc. You can only use constructs at the top level of a Jess program -- i.e., you can't use a construct on the right hand side of a rule, or inside a function.

A construct is basically the same as a *special form* in Lisp or Scheme. The odd syntax has to be handled specially by the parser, hence the name.

17.1. deffacts

Syntax:

```
(deffacts deffacts-name
    ["Documentation comment"]
    fact* )
```

Description:

A *deffacts* is just a list of facts. When the "reset" function is called, Jess clears working memory, asserts the special "initial-fact" fact, and then asserts all the facts defined by deffacts.

17.2. deffunction

Syntax:

```
(deffunction function-name (argument*)
    ["Documentation comment"]
    function call* )
```

Description:

A *deffunction* is a function written in the Jess language. You can call deffunctions from the Jess prompt, from a rule, or from another deffunction.

17.3. defglobal

Syntax:

```
(defglobal ?name = value
           [?name = value]* )
```

Description:

A *defglobal* construct defines one or more global variables and sets their initial values. **The name of a global variable must begin and end with an asterisk (*)**.

17.4. defmodule

Syntax:

```
(defmodule module-name
    ["Documentation comment"]
    [(declare (auto-focus value))]
    )
```

Description:

The *defmodule* construct introduces a Jess module. The current module is set to be the new module, so any rules defined after a defmodule will implicitly belong to that module. If a module has the `auto-focus` property, then all the rules in that module have the `auto-focus` property.

17.5. defquery

Syntax:

```
(defquery query-name
    ["Documentation comment"]
    [(declare (variables variable+)
              (node-index-hash value)
              (max-background-rules value))]
    conditional element* )
```

Description:

A *query* consists of an optional variable declaration followed by a list of *conditional elements.* A conditional element is either a *pattern*, or a grouping construct like "and", "or", or "not." Queries are used to search working memory for facts that satisfy the conditional elements. A query can be invoked using the "run-query*" function.

17.6. defrule

Syntax:

```
(defrule rule-name
    ["Documentation comment"]
    [(declare (salience value)
              (node-index-hash value)
              (auto-focus TRUE | FALSE)
              (no-loop TRUE | FALSE))]
    conditional element*
    =>
    function call* )
```

Description:

A *rule* consists of a left-hand side, the symbol "=>", and a right-hand side, in that order. The left-hand side is made up of zero or more *conditional elements*, while the right-hand side consists of zero or more function calls. A conditional element is either a *pattern*, or a grouping construct like "and", "or", or "not." The conditional elements are matched against Jess's working memory. When they all match, and if the engine is running, the code on the rule's right-hand side will be executed.

17.7. deftemplate

Syntax:

```
(deftemplate template-name
    [extends template-name]
    ["Documentation comment"]
    [(declare  (slot-specific TRUE | FALSE)
               (backchain-reactive TRUE | FALSE)
               (from-class class name)
               (include-variables TRUE | FALSE)
               (ordered TRUE | FALSE))]
```

```
     (slot | multislot slot-name
         ([(type ANY | INTEGER | FLOAT | NUMBER | SYMBOL | STRING |
                LEXEME | OBJECT | LONG)]
         [(default default value)]
         [(default-dynamic expression)]
         [(allowed-values expression+)])*)
```

Description:

A *deftemplate* describes a kind of fact, precisely in the same way as a Java class describes a kind of object. In particular, a deftemplate lists the "member variables" (called *slots*) that this particular kind of fact can have.

Deftemplate definitions can include a "declare" section. Each declaration will affect either how the template is defined or how facts that use the template will behave.

The first declaration we'll learn about is `slot-specific`. A template with this declaration will be matched in a special way: if a fact, created from such a template, which matches the left-hand-side of a rule is modified, the result depends on whether the modified slot is named in the pattern used to match the fact. As an example, consider the following:

```
Jess> (deftemplate D (declare (slot-specific TRUE)) (slot A) (slot B))

(defrule R
  ?d <- (D (A 1))
  =>
  (modify ?d (B 3)))
```

Without the "slot-specific" declaration, this rule would enter an endless loop, because it modifies a fact matched on the LHS in such a way that the modified fact will still match. With the declaration, it can simply fire once. This behavior is actually what many new users expect as the default, so the `slot-specific` declaration probably ought to be used most of the time.

The `from-class` declaration lets you derive a template from a Java class. By default, only JavaBeans properties generate slots. But if you include the optional `(include-variables TRUE)` declaration in your template, then Jess will also create slots corresponding to the public instance variables of the class.

Other declarations are described in this chapter.

The `backchain-reactive` and `slot-specific` declarations are inherited by child templates.

17. Jess Constructs

18. Jess – the Rule Engine - API

Jess - the Rule Engine for the Java Platform v71p2

This is the Java API documentation for the Jess rule engine.

See:
Description

The Jess Library	
jess	Implements the core of the Jess rule engine.
jess.awt	These event adapters let you run Jess code in response to Java GUI events.
jess.factory	Provides a hook to allow Jess extensions like FuzzyJess to pass additional data within the Rete network during pattern matching.
jess.jsr94	Implements the JSR94 or "javax.rules" API.
jess.server	Implements the Jess debug server.
jess.swing	Provides facilities for interacting with Swing GUIs from Jess.
jess.tools	
jess.xml	Provides facilities for reading and writing JessML, Jess's own XML rule language.

The JSR94 API	
javax.rules	The core of the `javax.rules` API.
javax.rules.admin	The administration component of the `javax.rules` API.

This is the Java API documentation for the Jess rule engine. The main package is "jess", and "jess.Rete" is the most important class.

Related Documentation

For overviews, tutorials, examples, guides, and tool documentation, please see:

- The Jess Home Page

19. The Rete Algorithm

19.1. Disclaimer

The information in this Section is provided for the curious reader. An understanding of the Rete algorithm may be helpful in planning rule-based systems; an understanding of Jess's implementation probably will not. Feel free to skip this section and come back to it some other time. You should not take advantage of many of the Java classes mentioned in this section. They are internal implementation details and any Java code you write which uses them may well break each time a new version of Jess is released.

19.2. The Problem

Jess is a rule engine. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements (*rules*) to a set of data (the *working memory*). You define the rules that make up your own particular rule-based system. Jess rules look something like this:

```
Jess> (defrule library-rule-1
    (book (name ?X) (status late) (borrower ?Y))
    (borrower (name ?Y) (address ?Z))
   =>
    (send-late-notice ?X ?Y ?Z))
```

This rule might be translated into pseudo-English as follows:

```
Library rule #1:
If
  a late book exists, with name X, borrowed by someone named Y
and
  that borrower's address is known to be Z
then
  send a late notice to Y at Z about the book X.
```

The book and borrower entities would be found on the working memory. The working memory is therefore a kind of database of bits of factual knowledge about the world. The attributes (called *slots*) that things like books and borrowers are allowed to have are defined in statements called `deftemplates`. Actions like `send-late-notice` can be defined in user-written functions in the Jess language (`deffunction`s) or in Java (`Userfunction`s) For more information about rule syntax refer to the chapter "Making your own Rules."

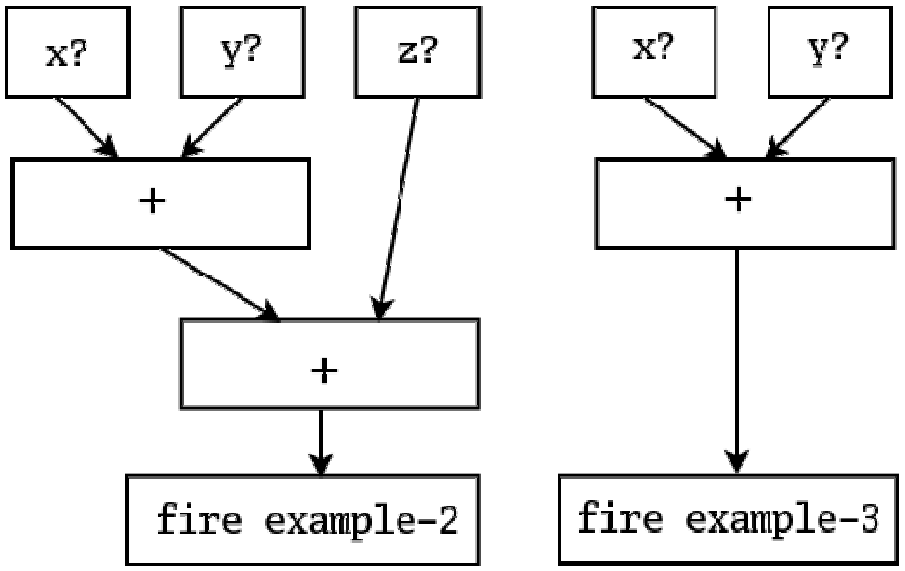
The typical rule-based program has a fixed set of rules while the working memory changes continuously. However, it is an empirical fact that, in most rule-based programs, much of the working memory is also fairly fixed from one rule operation to the next. Athough new facts arrive and old ones are removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for the rule engine is very inefficient. This obvious implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the working memory and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the working memory is stable, most of the tests will be repeated. You might call this the *rules finding facts* approach and its computational complexity is of the order of O(RF^P), where R is the number of rules, P is the average number of patterns per rule LHS, and F is the number of facts on the working memory. This escalates dramatically as the number of patterns per rule increases.

19.3. The Solution

Jess instead uses a very efficient method known as the Rete (Latin for *net*) algorithm. The classic paper on the Rete algorithm *("Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19 (1982), 17-37)* became the basis for a whole generation of fast rule engines: OPS5, its descendant ART, CLIPS, and of course Jess. In the Rete algorithm, the inefficiency described above is alleviated (conceptually) by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like O(RFP), or linear in the size of working memory. Our discussion of the Rete algorithm is necessarily brief. The interested reader is referred to the Forgy paper or to *Giarratano and Riley, "Expert Systems: Principles and Programming", Second Edition, PWS Publishing (Boston, 1993)* for a more detailed treatment.
The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS. Facts that are being added to or removed from the working memory are processed by this network of nodes. At the bottom of the network are nodes representing individual rules. When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an *activation*. The associated rule may have its RHS executed (*fired*) if the activation is not invalidated first by the removal of one or more facts from its activation set.
Within the network itself there are broadly two kinds of nodes: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function. Subtypes of these two classes of node are also used and there are also auxilliary types such as the terminal nodes mentioned above.
An example is often useful at this point. The following rules:

```
(defrule example-2      (defrule example-3
    (x)                     (x)
    (y)                     (y)
    (z)                     => )
    => )
```

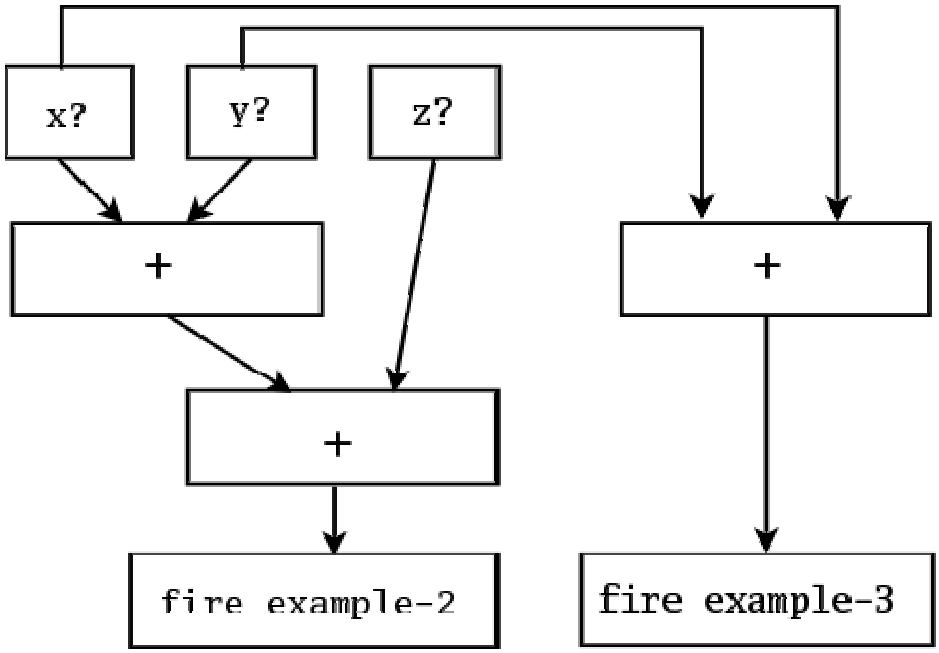
might be compiled into the following network:



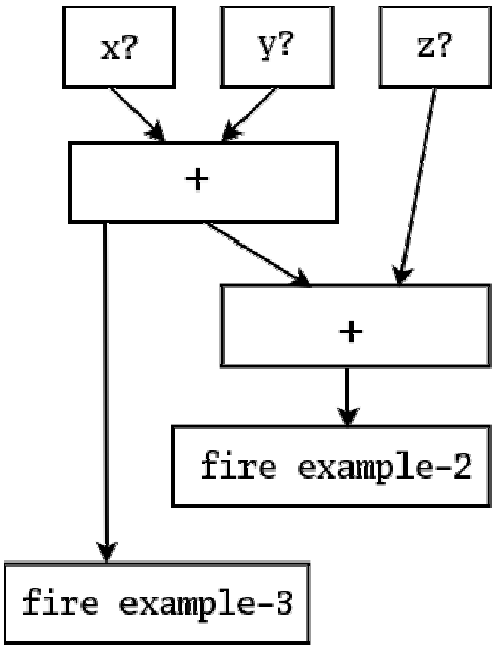
The nodes marked x?, etc., test if a fact contains the given data, while the nodes marked + remember all facts and fire whenever they've received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network as they added to the working memory. Each node takes input from the top and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and, if the test passes, sends the fact downward to the next node. If the test fails, the one-input nodes simply do nothing. The two-input nodes have to integrate facts from their left and right inputs, and in support of this, their behavior must be more complex. First, note that any facts that reach the top of a two-input node could potentially contribute to an activation: they pass all tests that can be applied to single facts. The two input nodes therefore must remember all facts that are presented to them, and attempt to group facts arriving on their left inputs with facts arriving on their right inputs to make up complete activation sets. A two-input node therefore has a *left memory* and a *right memory*. It is here in these memories that the inefficiency described above is avoided. A convenient distinction is to divide the network into two logical components: the single-input nodes comprise the *pattern network*, while the two-input nodes make up the *join network*.

19.4. Optimizations

There are two simple optimizations that can make Rete even better, The first is to share nodes in the pattern network. In the network above, there are five nodes across the top, although only three are distinct. We can modify the network to share these nodes across the two rules (the arrows coming out of the top of the x? and y? nodes are outputs):



But that's not all the redundancy in the original network. Now we see that there is one join node that is performing exactly the same function (integrating x,y pairs) in both rules, and we can share that also:



The pattern and join networks are collectively only half the size they were originally. This kind of sharing comes up very frequently in real systems and is a significant performance booster! You can see the amount of sharing in a Jess network by using the `watch compilations` command. When a rule is compiled and this command has been previously executed, Jess prints a string of characters something like this, which is the actual output from compiling rule example-2, above:

```
example-2: +1+1+1+1+1+1+2+2+t
```

Each time `+1` appears in this string, a new one-input node is created. `+2` indicates a new two-input node. Now watch what happens when we compile example-3:

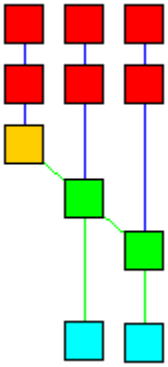
```
example-3: =1=1=1=1=2+t
```

Here we see that `=1` is printed whenever a pre-existing one-input node is shared; `=2` is printed when a two-input node is shared. `+t` represents the terminal nodes being created. (Note that the number of single-input nodes is larger than expected. Jess creates separate nodes that test for the head of each pattern and its length, rather than doing both of these tests in one node, as we implicitly do in our graphical example.) No new nodes are created for rule example-3. Jess shares existing nodes very efficiently in this case.

19.5. Implementation

Jess's Rete implementation is very literal. Different types of network nodes are represented by various subclasses of the Java class `jess.Node`: `Node1`, `Node2`, `NodeNot2`, `NodeJoin`, and `NodeTerm`. The `Node1` class is further specialized because it contains a *command* member which causes it to act differently depending on the tests or functions it needs to perform. For example, there are specializations of `Node1` which test the first field (called the *head*) of a fact, test the number of fields of a fact, test single slots within a fact, and compare two slots within a fact. There are further variations which participate in the handling of multifields and multislots. The Jess language code is parsed by the class `jess.Jesp`, while the actual network is assembled by code in the class `jess.ReteCompiler`. The execution of the network is handled by the class `Rete`. The `jess.Main` class itself is really just a small demonstration driver for the jess package, in which all of the interesting work is done.

The `view` command is a graphical viewer for the Rete network itself; I have used this as a debugging tool for Jess, but it may have educational value for others, and it may help you to design more efficient systems of rules in Jess. Issuing the `view` command after entering the rules example-2 and example-3 produces a very good facsimile of the drawing (although it correctly shows the larger number of one-input nodes). The various nodes are color-coded according to their roles in the network; `Node1` nodes are red; `Node2` nodes are green; `NodeNot2` nodes are yellow; and `Defrule` nodes are blue. The orange node in the figure is a "right-to-left adapter" node; one of these is always used to connect the first pattern on a rule's LHS to the network. Passing the mouse over a node displays information about the node and the tests it contains; double-clicking on a node brings up a dialog box containing the same information (for join nodes, the memory contents are also displayed, while for `Defrule` nodes, a pretty-print representation of the rule is shown). See the description of the `view` function for important information before using it.



19.6. Efficiency of rule-based systems

Jess's rule engine uses an improved form of a well-known algorithm called Rete (Latin for "net") to match rules against the working memory. Jess is actually faster than some popular rule engines written in C, especially on large problems, where performance is dominated by algorithm quality.

Note that Rete is an algorithm that explicitly trades space for speed, so Jess' memory usage is not inconsiderable. Jess does contain some commands which will allow you to sacrifice some performance to decrease memory usage. Nevertheless, Jess' memory usage is not ridiculous, and moderate-sized programs will fit easily into Java's default 16M heap.

The single biggest determinant of Jess performance is the number of *partial matches* generated by your rules. You should always try to obey the following (sometimes contradictory) guidelines while writing your rules:

- Put the *most specific* patterns near the top of each rule's LHS.
- Put the patterns that will match the *fewest facts* near the top of each rule's LHS.
- Put the *most transient* patterns (those that will match facts that are frequently retracted and asserted) near the bottom of a LHS.

You can use the `view` command to find out how many partial matches your rules generate. See this chapter on How Jess Works for more details.

19.6.1. Sun's HotSpot Virtual Machine

Because Jess is a memory-intensive application, its performance is sensitive to the behavior of the Java garbage collector. Recent JVMs from Sun feature an advanced Java runtime called HotSpot which includes a flexible, configurable garbage collection subsystem. Excellent articles on GC performance tuning are available at Sun's web site. Although every Jess rule base is different, in general, Jess will benefit if the heap size and the object nursery size are each set larger than the default. For example, on my machine, Jess' performance on the Miranker *manners* benchmark with 90 guests is improved by 25% by increasing the initial heap size and

nursery size to 32 and 16 megabytes, respectively, from their defaults of 16 meg and 640K. You can do this using

```
java -XX:NewSize=16m -Xms32m -Xmx32m jess.Main <scriptfile>
```

Note that the object nursery is a subset of the Java heap set aside for recently-allocated objects; the total heap size in this example is 32M, not 48M.

20. For More Information...

20.1. ... about Jess

- Friedman-Hill, Ernest, "Jess in Action: Rule-based Systems in Java", ISBN 1930110898.
- The Jess FAQ at http://www.jessrules.com/FAQ.shtml includes the solutions to some common problems.
- The jess-users mailing list.

  There is a moderated Jess email discussion list you can join. This is the best place to get your Jess questions answered quickly. To get information about the jess-users list, send a message to majordomo@sandia.gov containing the text

```
help
info jess-users
end
```

  as the body of the message. There is an archive of the list at http://www.mail-archive.com/jess-users@sandia.gov.

- Bug reports.

  No software is ever perfect. Despite our best efforts and extensive testing, there may still be bugs in Jess. Please read the release notes for specific information. Comments and bug reports are welcome. Contact me at `ejfried@sandia.gov` so I can fix them for a later release.

20.2. ... about Java and Java Programming

- Java's home page
- The JavaRanch forums, in my opinion the best place on Earth to ask Java questions.

20.3. ... about Rule Engines and Expert Systems

- Giarratano and Riley, "Expert Systems: Principles and Programming", Second Edition; ISBN 0878353356.
- Stuart Russell and Peter Norvig, "Artificial Intelligence - A Modern Approach," ISBN 0131038052.
- John Durkin, "Expert Systems - Design and Development," ISBN 0023309709.
- Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, Nigel Shadbolt, Walter Van de Velde and Bob Wielinga, "Knowledge Engineering and Management - The Common KADS Methodology," ISBN 0262193000.
- Nils J. Nilsson, "Principles of Artificial Intelligence," ISBN 0934613109.
- Jay Aronson and Efraim Turban, "Decision Support and Intelligent Systems", ISBN 0137409370.
- Mark Watson, "Intelligent Java Applications for the Internet and Intranets"

20. For More Information...

21. Release Notes

Jess 7.1 introduces some new features on top of the foundation laid in Jess 7.0.

21.1. New features in Jess 7.1

Templates inherit declarables from parent. In particular, templates inherit backwards chaining reactivity and slot-specific behavior from their parent template (if they have one.)

Test CEs allowed in logical. You're now allowed to use `test` inside of the `logical` CE.

Alternative character sets. The `batch` function and the `jess.Batch` class's various methods can accept a charset argument, so Jess can read files written in arbitrary character sets.

Test CE folding. The `test` conditional element is now more efficient -- in fact, its performance is equivalent to declaring the same tests somewhere in the preceding pattern. A `test` pattern is no longer compiled to a real join node in the Rete network. You can now use the `test` CE anywhere you want to improve program readability or in generated code, without penalty.

Automatic strength reduction. Many uses of the "eq" and "neq" functions in the patterns of a rule will be automatically converted to more efficient direct matching. Although direct matching is still to be preferred because it is less verbose, generated code can use "eq" and "neq" uniformly with other functions and Jess will do this optimization when possible.

Dot notation. The "simplified" or "Java" patterns can now use a "dot notation" to reference slots in other patterns. This powerful capability greatly expands the usefulness of Java patterns and makes auto-generating Jess code much easier. Futhermore, The notation "?x.y" in procedural code will be interpreted as "slot y of the fact in x".

Name restrictions. Slot names and variable names may not contain a period ('.').

Variables not bound by Java patterns. The semantics of Java patterns, introduced in Jess 7.0, have changed slightly. These patterns no longer bind user-accessible variables to any slots they're used in. To access slot values of facts matched by Java patterns, use the `fact-slot-value` function or the `jess.Fact.getSlotValue(java.lang.String)` method.

allowed-values. Template slots can now use the allowed-values qualifier, which states which discrete values a slot can hold. Jess will verify that your code obeys this statically; optionally, it can also be verified at runtime. Using allowed-values where appropriate will take advantage of some new optimizations to be implemented in Jess 7.1.

Peering of Rete objects. `jess.Rete` objects can share their rule networks, resulting in reduced memory footprint and faster setup times when you're using multiple engines in a single application.

elif. The `if` function supports `elif` blocks.

ValueFactory. Jess uses a pooling factory for Value objects, resulting in generally lower memory consumption if you parse a lot of data. You can use the same factory for the Value object you create. If you modify your code to use it, you can conserve memory since ValueFactory caches some of the Values it creates. HashCodeComputer, the class that Jess uses to keep track of which objects a "value objects", is also public.

break. The `break` function lets you terminate loops early as well as return from the actions of a rule without popping the focus stack.

Removing constructs. The `jess.Rete` class now includes methods for removing modules, templates, globals, and all other constructs.

21.2. New features in Jess 7.0

Jess 7 has many new features. The manual has had a major rewrite for this version, although some new features may be underdocumented. These notes are intended to help with that.

New features in Jess 7 include, in no particular order:

Java patterns. You can use a new, simplified syntax for pattern matching that looks a lot like Java Boolean expressions. You can read about it here.

Static imports. Importing a class now causes all its static members to be defined as functions. See here for details about this new and useful feature!

Lambda expressions. You can now create unnamed functions and easily use them to implement Java interfaces. See here for details.

New default for `definstance`. The definstance function's default behavior is now to use PropertyChangeEvents whenever they are available. It's generally not necessary to use the "static" or "dynamic" qualifiers anymore, unless you want to force Jess not to use a PropertyChangeListener even when that facility is available. You can explicitly use a qualifier of "auto" as a synonym for the default.

Eclipse-based IDE. Charlemagne includes a rule development environment called "JessDE" based on the Eclipse open-source IDE. The toolset includes an editor, a debugger, and a host of small tools to help with rule development and deployment. You can read more about it here.

Simplified defquery syntax. A new `run-query*` function brings a JDBC-like interface to working memory queries. Read about it here.

The "slot-specific" declaration for templates. Deftemplate definitions can now include a "declare" section just as defrules can. There are several different properties that can be declared. One is "slot-specific". Changes to facts from slot-specific templates won't trigger rule activations unless the modified slot is actually mentioned on the LHS of the rule.

The "backchain-reactive" declaration for templates. This declaration lets you declare that a template is backward chaining reactive when you create it, rather than after the fact.

The "from-class" declaration for templates. This provides a different syntax for defining templates based on Java classes. Using "from-class", you can apply all the other template declarations to shadow fact templates.

Public member variables as slots. When you define a template using the deftemplate from-class syntax or using the Rete.defclass() method, you can tell Jess to include public member variables as slots in the template. Do this by passing "true" as the optional fourth argument to Rete.defclass(), or by including the declaration "(include-variables TRUE)" in your deftemplate. There is not yet a way to turn this feature on when using the defclass function.

The "no-loop" declaration for rules. If you use `(declare (no-loop TRUE))`, then nothing that a rule does while firing can cause the immediate reactivation of the same rule; i.e., if a no-loop rule matches a fact, and the rule modifies that same fact such that the fact still matches, the rule will not be put back on the agenda, avoiding an infinite loop.

Matching with regular expressions, and the `regexp` function. Under JDK 1.4 and up, Jess 7 has regular expressions support. You can directly match any field with a regular expression.

The -stacktrace switch. The error messages printed by the `jess.Main` class have always (deliberately) included a stack trace to help you find the specific part of Jess that rejected your input. Unfortunately, stack traces are frightening to many non-programmers. Furthermore, some people assume that when they see a stack trace, they're seeing a bug in Jess. Therefore, `jess.Main`'s new default behavior is not to display a stack trace on error. To get the old behavior, you can include the `-stacktrace` command-line switch when you start `jess.Main`.

RU.LONG is now a first-class type. You can write long literals by appending an "L" to an integral value. Values of "long" type can now be used in arithmetic and logical expressions. The `long` function still exists but, as it's no longer needed, it is deprecated.

The "forall" conditional element. Jess now includes a "forall" conditional element. The "forall" grouping CE matches if, for every match of the first pattern inside it, all the subsequent patterns match. See here for more information.

The "accumulate" conditional element. The "accumulate" CE is lets you count facts, add up fields, store data into collections, etc, all as a part of pattern matching. See here for more information.

Subrules no longer user-visible. The "or" CE causes rules that use it to be broken into subrules, as described in the manual. However, the individual subrules will no longer be visible to the user; they're an implementation detail that should not concern most programmers.

Native XML parser. Jess 7 defines its own XML rule format, *JessML*, designed so that it's easy for the language to support Jess's special features. An XML Schema file describing this format is included in the distribution. We hope to eventually provide XSLT scripts to transform this language into RuleML and other standard rule languages. Several examples of the XML format are included in the examples directory which are exact machine translations of standard Jess examples. The `batch` function knows how to read these XML files in, and there are a number of useful public Java classes in the package `jess.xml` for working with this new format, including a tool to translate any Jess code into JessML.

Public API for rule creation. The XML parser is defined in its own package, and works by creating jess.Defrule objects using the Java API. This is an existence proof that such a thing is now possible. Documentation on *how* is forthcoming.

Better handling for objects with mutable hashCodes. Object identity in Java is a slippery thing, as experienced programmers know; the interpretation of identity and equality can be intimately tied to the class-specific implementations of the `equals` and `hashCode` methods. Jess has historically had some problems with Java objects whose hash code changes over time. These problems have been resolved in Jess 7. Objects with changing hashCodes can be safely added to working memory even in situations that have caused problems in the past. See "Java objects in working memory" for details.

Better reflection performance. Jess's performance while scripting Java objects in procedural code has improved by 20-50% in this release, especially in loops.

Better method names. The `RU.EXTERNAL_ADDRESS` constant, the `jess.Value.externalAddressValue()` method, and the `external-addressp` function are all deprecated in favor of the new equivalents `RU.JAVA_OBJECT`, `jess.Value.javaObjectValue()`, and `java-objectp`.

New overloaded add() methods. The `jess.ValueVector` class has a new set of overloaded `add()` methods that make Java code that uses Jess lists look cleaner.

CLEAR. One minor but potentially important behavioral change in this release is that the `jess.Rete.clear()` method doesn't modify the event mask, set of event listeners, or watch state. Although this is a break with the traditional semantics of the clear() method, it makes more sense, especially for embedded applications.

21.3. Porting from Jess 7

Jess 7 code will generally run fine in Jess 7.1, with a few small exceptions. One issue is that the period character '.' is no longer valid in variable or slot names, since the period is now used as a "dot operator" to denote membership. If this breaks existing code, you can choose to use some other character (we suggest the octothorpe '#') as a membership operator by adding the following to the beginning of your Jess code:

```
((engine) setMemberChar #)
```

21.4. Porting from Jess 6

In general, Jess 6 applications will run unchanged (either Java API code, or Jess language code,) with a few exceptions, as noted below. Nevertheless, you may want to change your application to take advantage of some of the newer features -- for example, the new `run-query*` method.

Restrictions on variable names. Jess 7 accepts a more limited syntax for variable names than did earlier versions of Jess. Only letters, numbers, dash, asterisk, colon, and underscore are

legal characters in variable names in this release; in particular, the period (.) character is no longer allowed as part of a variable name. Hopefully this won't affect any existing code.

Newly deprecated methods. The `Rete.executeCommand()` method is deprecated in favor of the new `eval()` method.

Previously deprecated methods removed. All deprecated methods from Jess 6 have been removed from Jess 7. In particular, the "assert()" method from the Rete class has been replaced with "assertFact()".

New deprecated functions. The Jess function "multifieldp" is now deprecated; use "listp" instead. The `RU.EXTERNAL_ADDRESS` constant, the `jess.Value.externalAddressValue()` method, and the `external-addressp` function are all deprecated in favor of the new equivalents `RU.JAVA_OBJECT`, `jess.Value.javaObjectValue()`, and `java-objectp`.

USERFUNCTION_CALLED events. The data associated with a JessEvent of type USERFUNCTION_CALLED is no longer a Userfunction object; it's now a Funcall object.

Context.setVariable() behavior changed. The setVariable() method of jess.Context behaves a little differently now. In general, you won't notice a change unless you were using variables defined at the command prompt as if they were global variables. They never were, and never acted as if they were -- and they're even less so now.

Arguments to "batch". The path argument to the Jess function "batch" must be a double-quoted string; using an unquoted symbol instead will lead to undefined behavior.

Value objects. Jess now assumes that all objects (except for Collections) are value objects by default. If you're working with a class that is *not* a value class, it's very important that you tell Jess about it by using the `set-value-class` function. See "Java objects in working memory" for details.

Return types. A very few public methods that used to return concrete collection classes are now declared to return the corresponding interface; for example, Rete.getSupportingTokens() now returns List rather than ArrayList.

21. Release Notes

22. Change History

Version 7.1p2 (November 5th, 2008):

QueryResult distinguishes between "no results" and "before first result" states (thanks Bob Kirby). Fixed a slot-specific bug and a no-loop bug (thanks Neal Wyse.) Fixed a problem with salience evaluation in peered engines (thanks George Williamson).

Version 7.1p1 (August 6th, 2008):

Small documentation fixes; added Rete.setMemberChar() method; fix bug in overriding default slot values in extending templates.

Version 7.1 (July 8th, 2008):

Some small documentation fixes; avoid an erroneous error message (thanks Bob Kirby); a better error message for pretty-printing a Java function (thanks Wolfgang Laun);

Version 7.1RC1 (June 2nd, 2008):

Exceptions include source file names. Exceptions from "batch" include the line number in the batch file, not the caller. ValueFactory is public and documented. "<>" behaves properly for more than 2 arguments (thanks Henrique Lopes Cardoso). Added uniform methods for removing all types of constructs. Jess can understand that "<Fact-6>" is a reference to the fact with ID 6, in many contexts. HashCodeComputer is public. JessDE warns when rules, queries, deffunctions are redefined. Modules can be declared "auto-focus". "call" will invoke a static method on a named class in preference to a method of java.lang.String if the target is an RU.SYMBOL. RU.valueToObject will convert a LIST to an Object as an Object[]. Rete uses system property to find scriptlib. Template properties (backchain-reactive, slot-specific) are inherited. clear() doesn't touch event handlers, mask, or watch state. Parser handles dotted vars in direct LHS tests. "get" fetches slot values from Fact objects. "(read)" and "(explode$)" handle nested variables.

Version 7.1b3 (April 2nd, 2008):

Line numbers in JessDE problem markers are one-based (thanks Win Carus). Reject "if" without "then" in all cases (thanks Michael Atighetchi). JessDE understands Eclipse projects that store files outside the workspace. The "(add)" function respects logical dependencies (thanks Lorie Ingraham). Fixed a bug with slot-specific and multiple modifies (thanks Florian Fischer). The static constraint checker will ignore variables as slot values (thanks Wolfgang Laun.)

Version 7.1b2 (February 20th, 2008):

Added "continue" (thanks Win Carus). get/set delegate to get-member/set-member on failure. Fixed set-strategy bug (thanks Neal Wyse). FactIdValue doesn't cache fact id (thanks Art Griesser). no-loop is thread-specific (thanks Yixi Chen). Added Rete.createPeer() method (thanks John Adair.)

Version 7.1b1 (November 16th, 2007):

Fix bug attempting to resolve blank variables (thanks Bob Kirby.) Optization: cull rule binding table at compile time. Allow 'test' CE inside 'logical' (thanks Denis Berthier). Deprecate Activation.isInactive() (has returned only 'false' for several releases.) Fixed bug wherein activations of deleted rules could persist (thanks Florian Fischer). "batch" can accept a charset. Fix bug with "slot-specific" and "not" (Fischer.) Dot notation for Bean properties of Java objects. Added "Rete.hasActivations()". Fix concurrency issues with peered Retes during "clear()."

Version 7.0p2 (October 21st, 2007):

Fix agenda bug (thanks Brian Rogosky).

Version 7.1a3 (October 3rd, 2007):

Keyed storage bsaved/bloaded (thanks Henrique Lopes Cardoso.) Don't obfuscate ClassResearcher.Property in trial version (thanks Wolfgang Laun). DEFMODULE

events. Dot notation in Java patterns. Fix bug in redefining templates which extend other templates. Fix logical bug (thanks Aaron Novstrup). Fix agenda bug (thanks Brian Rogosky.) Fix and/not bug (thanks Florian Fischer and Bob Kirby). Fold "test" CEs into preceding pattern. Translate many 'eq', 'neq' funcalls into direct matches.

Version 7.1a2 (July 19th, 2007):

Fix bug using "!=" and "<>" in compound infix expressions (thanks Richard Long.) Fix bug in run-query* with 'or' CE (thanks Robert Kirby.) Implement "allowed-values" slot qualifier. "return" at prompt now acts like "break" (thanks Wolfgang Laun). Lots of documentation fixes (Laun). Fixed a race with definstances in "reset". Faster startup: no AWT classes loaded by default. Fixed an issue with waitForActivations (thanks Chad Loder.) Fix an incremental reset bug with "not" (thanks Florian Fischer.) Fix bug in resolving redefined Userfunctions (thanks Henrique Lopes Cardoso). Queries work with peering (thanks Chad Loder.) Add Rete.listDeffunctions() method (Laun). "call" knows about lambdas.

Version 7.1a1 (May 14th, 2007):

Can pass the STRING "nil" to a Java method (thanks Win Carus). Added "elif" to "if" function. ValueFactory class with pooling for symbols. Fixed incremental reset bug (thanks Rand Waltzman). jess/RegexpMatch is Serializable (thanks Shyamal Pandya). Did a general sweep for concurrency issues. Fix JessML docs (thanks Henrique Lopes Cardoso.) Peering of Rete objects. Added "break" function.

Version 7.0p1 (December 21st, 2006):

New commands in JessDE: select construct, pretty-print construct. Portable Mac keybindings. "Java patterns" (infix test patterns) can use grouping via parentheses. Multiline comments and strings don't disturb JessDE formatting or damage repairer. JessDE Rete view shows all OR branches, and node colors in trial version (thanks Mark Proctor). "watch rules" output goes to watch router (thanks Helge Hartmann.) Repeated negated unifications in a multifield are kept (thanks Gary Riley).

Version 7.0 (November 1st, 2006):

Fixed a race condition in module switching (thanks Scott Krasnigor). Fixed an issue with matching defglobals in "not" conditional elements (thanks Joshua Undesser). Fixed RuntimeException on parsing a bad rule (thanks James Lefeu).

Version 7.0RC3 (October 12th, 2006):

Eclipse 3.1 support.

Version 7.0RC2 (October 10th, 2006):

Improve documentation for bload/bsave (thanks Henrique Lopez Cardoso). Fix Rete.removeUserfunction() (Cardoso). Add FilteringIterator, documentation, ByModule filter, fix (facts) documentation (thanks Christoph Bussler). Fix performance issue in Context. Manual includes "pricing engine" domain code. Fix agenda ordering issue (thanks Heinrich Flaig). ppdeftemplate doesn't print slots for from-class templates. Fix three accumulate bugs (thanks Neal Wyse and Phil Varner). Fix debugger protocol issues on Windows. Fix debugger error when displaying slot contents of retracted fact. Fix debugger classpath issues on Windows.

Version 7.0RC1 (September 5th, 2006):

Fix NPE using defgobals with 'or' connectives (thanks Howard Greenblatt). Reject nested parens early (thanks Yuping He). backchain-reactive declaration works for ordered templates (He). Token.getTime() and Token.getTotalTime() are now public. Don't need to put jess.jar on build path of Jess projects. "Open file..." works. RuleExecutionSets in JSR-94 driver are cloned for each RuleSession. Fix problem debugging files starting with blank lines. save-facts-xml/load-facts for XML (thanks Stacy Lovell). New "remove" function retracts all facts from a given template. XMLPrinter can now be used programmatically. "batch" in JessDE uses classpath properly. Added

package versioning. lowcase, upcase return symbol for symbol arguments. XML schema included. Various small JessML changes. Fix ClassCastException in jess.Multimap.remove() (thanks Neal Wyse) 'type' qualifier for multislots (Greenblatt). Better error message for wrong toplevel XML element. Add "as-list" function.

Version 7.0b7 (May 11th, 2006):

Boolean comparison operators work on Comparables. Support for braces in editor. Enormous improvements in editor "content assist" features. Undo removes both characters of autoinserted parentheses. Added context help for completion proposals. Fixed bug in modifying multifield facts (thanks Brian Zhou). JessML includes docstrings for all constructs (thanks Erich Oliphant). JessML now has imperative features. 36% of application test suite now runs in XML mode. Can specify an XML file as argument on command line. Fix error switching to Debug Perspective.

Version 7.0b6 (March 13th, 2006):

Fix bug in (logical) when modifying working memory from nested scopes (thanks Yuri Gribov). Added "add" function. Added "Java patterns." Deprecated executeCommand() in favor of eval(). *Massive* manual rewrites. Another logical/slot-specific bug fixed (thanks Shan Ming Woo.) Rete member in ClassSource transient (thanks Jonathan Sewall.) Overloaded updateObject() accepting property name (for Lakshmi Vempati). Fix synchronization of "undefinstance *" (thanks Abdul Quddoos Khan). Slight change to "accumulate" semantics (thanks Shan Ming Woo.) Overloaded "add()" methods in ValueVector.

Version 7.0b5 (January 5th, 2006):

Don't "wrap" external JessExceptions in another JessException. Fix a logical/slot-specific interaction (thanks Shan Ming Woo). QueryResult gives better error messages for undefined variable parameters or for calling getXXX() before next(). Pretty-print zero-length multislot patterns correctly (thanks Jonathan Sewall). (map) handles multi-argument functions (thanks Yuri Gribov.) (watch) accepts multiple arguments. Bug in handling multiple multifields in slot-specific templates fixed (thanks Howard Greenblatt). QueryResult has full Javadoc. Added "ppdefrule *". Static imports handle overloaded methods. Fixed incremental reset for accumulate (thanks Roger Studner). Defadvice for nested function calls (thanks Ivan Op de Beeck.) undeffacts function added. JessDE parses functions for rule salience values (Gribov). XMLPrinter handles "or" conditional element properly. Reparsing bug with "accumulate" in the JessDE fixed (thanks Jason Morris.) "modify" now once again modifies multiple slots simultaneously. "max" and "min" return their arguments using their original data types. Debugger now correctly steps though router I/O statements.

Version 7.0b4 (November 1st, 2005):

Fix slot-specific behavior for "null" property names in change events (thanks Shan Ming Woo). Fix SAX parse error for facts with no slots (thanks Wang Dong.) Fixed definstance name resolution across modules (Woo). New "map", "filter", and "list" functions for functional programming. Run configurations are reused in Eclipse. Can pass command-line arguments from Eclipse Run dialog. Defglobals show up in JessDE debugger. Spaces in path names won't confuse jess.bat (thanks Eric Schubert.) Fix deadlock in debugger when stepping over (read) and (readline) calls (thanks Jim Goodwin.) "Start suspended" checkbox in debugger. Can specify jess.Main alternate in launch configuration. Tiny font in manual navbar fixed. New URL for Jikes in manual. New "quick start" section in manual, and a new simplified engine API as described therein: mark(), resetToMark(), add(), addAll(), getObjects().

Version 7.0b3 (September 15th, 2005):

Continued work on manual and javadocs. JSR94 driver available in demo jar. (help) gives help on constructs and is generally more helpful. list-functions$ doesn't list static

imports. Deprecate the term "external address" in favor of "Java object", and deprecate all names based on this term. Add the "auto" qualifier for definstance, and change the default behavior to "auto". Swing versions of Canvas, TextAreaWriter. Debugger handles spaces in path names. Reverse ill-considered "optimization" in NodeNot2 (thanks Yuri Gribov).

Version 7.0b2 (August 11th, 2005):

Fix interactions between code folding and auto-paren insertion (thanks Howard Greenblatt). Format in editor no longer appends a blank line to document. Lots of work on the manual and Javadocs. Fix a memory leak in TokenTree (thanks Steven Goncalo.) min and max work with LONG (thanks Shan Ming Woo.) Static imports! Fix for accumulate bug (Woo, again.) Definstance uses template resolution procedure as assert (Woo). Fixed an erroneous report of variable used before definition (thanks Bill Robinson).

Version 7.0b1 (July 11th, 2005):

JessDE editor has preference to turn on line numbering. Fixed another leak in (logical). Close "backdoor" for putting lists in single slots (thanks Timothy Redmond.) Better error messages for constructor failures. Per-slot logical dependencies. Defclass templates get proper slot types. Fix pretty-printing of slot types and error-reporting for non-symbolic slot types (thanks Howard Greenblatt.) XML representation for defclasses. All-new JSR94 driver. "Folding" in JessDE editor. Editor deals with bad build path (thanks Howard Greenblatt). Defclass names now properly "modular." Fixed problem unifying negated variables in nested groups (thanks Neal Wyse.) Undefrule removes activations regardless of current module (Neal Wyse.) Editor doesn't warn about recursive deffunction calls. Public member variables as slots. "div" handles multiple arguments (thanks Shan Ming Woo.)

Version 7.0a6 (March 23rd, 2005):

Fix incorrect error message in set/get-member functions. () syntax for empty list. Fix bug in compiling logical dependencies on backward-chaining triggers (thanks Timothy Redmond). Fix JDK version detection (thanks Bob Hablutzel.) New functions: "++", "--", "for". Debugger can both "step into" and "step over" (thanks Mitch Christensen). "require" and "provide". Default values for multislots parsed better (thanks Travis Nelson.) "Toggle comment" works better. Can map keys to JessDE editor commands. Editor recognizes set-current-module. Fix bug in JessDE editor autoedit strategy, inserting at end of all-comment buffer (Mitch Christensen.) Lots of new "quick fixes." Smarter completion processing. "Agenda" view in debugger. Source location always returns source files, never files in /bin (Mitch Christensen).

Version 7.0a5 (February 2nd, 2005):

New Rete network view in JessDE. Fixed problems inserting at end of document in JessDE editor. Improved autoformatting in JessDE editor. External Eclipse projects work fully. Runtime class loading errors reported as error markers. Improved content assist. Document change in "batch" behavior (thanks Jeff Pierce). Some better error messages. set-watch-router command (thanks G. Williams.) New "value object" facility; large speedups in Java-object-matching performance are possible. Slight improvement in auto-parenthesis insertion (thanks M. Christensen.) Added initial "quick fix" implementation to editor. Better error reporting for non-numeric arguments to match functions (thanks M. Christensen.)

Version 7.0a4 (December 7th, 2004):

Fixed infinite loop with outline view of empty file in JessDE. JessDE can find Java classes in more places. JessDE reports warnings on bad arguments to more functions. Content assist works for functions called at top level.

Version 7.0a3 (November 30th, 2004):

Fixed test CE in user-defined modules (thanks Sven Siorpaes). Fix NPE in modify. Fixed "Not a Not a list" error messages (thanks Joseph Tappero). Added multiline comments (for Ross Judson.) Added -debug switch to jess.Main. jess.Main no longer stops on the first error in a file. Fixed extra-newline problem in reading erroneous input interactively (thanks Maxim Tretyak). Eclipse debugger now included, based on new debug mode in jess.Main. Fix overeager query analyser (thanks W. Robinson.) Vastly improved editor performance. Editor finds classes in other projects and outside workspace. No more duplicate "Source" menu in JessDE (thanks Jason Morris.) Editor checks load-function calls. Fixed NPE when modifying Fact with non-negative not in working memory (thanks Adam Velardo.) Fixed NPE when extending a nonexistent deftemplate (thanks Rodolfo Martin). "Run as..." in Eclipse. Better parsing of doubles. jess.Context.setVariable only sets values locally. Editor uses Rete import table to resolve classes.

Version 7.0a2 (October 8th, 2004):

Problem calling some Java methods fixed (thanks Alan Moore.) Fix function index in manual (Jason Morris.) More documentation. Fixed spurious error reporting of variable use in defquery (William N. Robinson). Jess 7 works under JDK 1.3 (Alan Moore). Parser clearly reports misused constructs (Mong-Thao La). JessDE editor responds to resource changes (Howard Greenblatt). JessDE editor uses project build path for class loading (Mitch Christensen). Functions that list things in a module tell you what module they've been applied to. Fixed bug in slot-specific (thanks Dan Malka). fact-slot-value calls Fact.getIcon first. Merge readline fix from 6.1p6.

Version 7.0a1 (September 17th, 2004):

Unify "batch" and "batch-xml". Lots of improvements to manual (thanks Win Carus.) Funcall.toString() is more forgiving about arguments to "modify". (thanks Chris Huang.) Folded in changes from 6.1p7. Report undefined variables used in predicate constraints (thanks Mitch Christensen.) Fixed backward chaining triggers inside logical CE (thanks Steffen Luypaert). Parser responds sooner to erroneous input. Slot names in "modify" and "duplicate" can be variables. Faster reflective method calls, especially for member functions of java.lang.String. Better pretty-printing. Help command. fact-slot-value moved from scriptlib to Java. "ordered" declaration for deftemplates.

Version 7.0pre-a1 (March 1st, 2004):

Remove deprecated methods. Added "no-loop" rule declaration. Added "slot-specific", "from-class", and "backchain-reactive" template declarations. Added "regex" function, and regex matching using "/xxx/". Added -stacktrace switch to main. Better error messages! LONG is now a first-class type. Better handling for Java objects with mutable hashCodes. 20-40% performance boost for Java object manipulation. "forall" conditional element added. "accumulate" conditional element added. XML parser now supported. Public API for rule creation! Subrules now no longer user-visisble, except through getNext() and getConditionalElements(). Began manual rewrite.

Version 6.1p8 (June 7th, 2004):

Fixed undefrule leaving behind activations (thanks Neil Wyse).

Version 6.1p7 (May 7th, 2004):

Fixed deadlock problem (thanks Paul Kaib). Better performance with huge numbers of templates (thanks Travis Nelson.) Fix NPE when parsing empty (not) patterns. Fix PrettyPrinter bug with (not) and pattern bindings.

Version 6.1p6 (September 4th, 2003):

Fixed two incremental reset problems (thanks Kevin Kusy and Xiaocheng Luan.) str-cat was evaluating the first argument twice (thanks Neal Wyse.) Fixed bug using '|' constraint on unbound slots inside (logical) (thanks 'Ray'). (readline) now works better in immediate mode. Fixed query name resolution issue (thanks Felix Bachmann). "crlf" translates to 0x0D0A on Windows (thanks Morten Lind.)

Version 6.1p5 (September 4th, 2003):

Jess now works as a standard extension, including loading user classes (thanks Ted Neward). Fixed a potential stack overflow in FactList.processPendingFacts() (thanks Ray Mekert.) Error messages from jess.awt.Canvas.paint() are flushed (thanks Isilyn Cabarrubias).

Version 6.1p4 (July 8th, 2003):

Fix a problem with variable binding (thanks Kevin Kusy.)

Version 6.1p3 (June 26th, 2003):

Fix a problem with side-effects executing multiple times while resolving overloaded methods (thanks Chad Loder.) Fix an incremental reset bug (thanks Maxim Tretyak.) Binary version works with Java 1.2.2 and above.

Version 6.1p2 (May 21th, 2003):

Fix a problem with clearing defglobals (thanks Bob Orchard.) (reset) removes all top-level variables, even "fake defglobals." Some improved error reporting. Public Test1.getMultiSlotIndex() method.

Version 6.1p1 (May 6th, 2003):

Fix a problem with pretty-printing of defqueries. Fix a (not) CE bug (thanks Olga Medvedeva). Don't call run(0) from run-defquery (thanks Glenn Williams.)

Version 6.1 (April 9th, 2003):

No nontrivial changes.

Version 6.1RC1 (March 24th, 2003):

Fixed another leak and another deadlock in "logical" (thanks Glenn Williams.) Reflection machinery now tries more aggressively to convert between floats and RU.FLOATs, ints and RU.INTEGERs, etc, in both directions. assert-string just returns FALSE on duplicate fact (thanks Thomas Diesler.) Better system for renaming variables in nested NOTs. Fix a race condition in JessEventSupport (thanks Chad Loder.) Added listDefclasses, etc. Fixed "breadth" conflict resolution strategy.

Version 6.1b3 (February 28th, 2003):

Most of the Rete "listXXX" methods lock and copy to prevent ConcurrentModificationExceptions. Fix problem with triply nested (not) (thanks Alan Moore.) Fix a race condition with modify and run-until-halt (Mr. Moore, again.) "unique" CE is now a no-op. Deadlocks, memory leaks, and other issues with the "logical" CE repaired (many thanks to Travis Nelson and Glenn Williams.)

Version 6.1b2 (February 14th, 2003):

Proper handling for "OR" CE's deeply nested inside NOTs. Nested, negated "test" CEs now work properly. Add "getThisActivation" and "getThisRuleName" to Rete (thanks Richard Kasperowski.) Can test fact bindings against previously defined variables. Simple public ConditionalElement class. "exists" handled by parser.

Version 6.1b1 (January 28th, 2003):

Code cleanup: remove unused return value from callNodeLeft/Right, add Context argument to callNodeLeft/Right, and remove ThreadLocal Rete.s_engines and static Rete.getEngine() method. When any fact, shadow or not, is modified, either from Jess, from the Java API, or from a Bean event, a single FACT | MODIFIED event will be generated. Many of the built-in package classes are no longer public. Added Rete.updateObject and (update). Added Rete.modify(). Added Rete.watch(), unwatch(). (watch facts) reports modify again. max-background-rules declaration added to defquery. Javadoc coverage improved. Fixed ppdeftemplate handling of default-dynamic (thanks Ross Judson.)

Version 6.1a5 (January 15th, 2003):

Fixed a bug with direct matching of fact bindings (thanks Morten Vigel). Fixed a typo in Node1TNEV1 (Ralph Grove.) Added "synchronized" and "get-strategy" functions. Calling

"set-strategy" now changes the default strategy for new modules. Fixed a deadlock in defquery (thanks Chad Loder.) Public interface "Test" renamed to "TestBase." Fixed bug in pretty-printing of deftemplates. Performance is back up to 6.0 equivalent.

Version 6.1a4 (August 30th, 2002):

Fixed a 6.1a3 bug where negated patterns could trigger backward chaining. "eval" and "build" can use variables from the calling context. Clarify and extend "app object" mechanism; Jess can be installed as a standard extension. Partially ameliorate performance regression from 6.1a3.

Version 6.1a3 (July 23rd, 2002):

Fixed a bug in save-facts (thanks Scott Trackman). Fixed deadlock problem in retract/undefinstance (thanks Blaine Bell.) Mihai Barbuceanu helped modernize the "bag" command. Various API enhancements. New Rete constructor with "app-object" argument. Massive refactoring of rule compiler to handle arbitrarily nested AND and NOT CEs in a new, simpler way; universal quantifiers are now handled correctly. Fixed a bug with the logical CE when adding new rules to a running system. Undefinstanced, nonserializable Beans now no longer prevent Rete serialization (thanks James Gallogly and Alan Moore.)

Version 6.1a2 (May 22st, 2002):

Fixed early return bug in try/catch/finally (thanks Chad Loder). (modify) now immediately modifies shadow fact for static definstances. A shadow fact can tell you the category of definstance it belongs to. New version of `format` function now has a %n conversion specifier, and also fixed an infinite loop when formatting floating-point zeroes. Refactoring in logical -- should not be user visible.

Version 6.1a1 (April 3rd, 2002):

Class import table now public. "view" cmmand improved, and bugs fixed. Activations now have individual salience values. Many methods in jess.Context now public. Rete.retract() now calls undefinstance() if needed. Rete.undefinstance() returns shadow fact. Matching against defglobal works (thanks Simon Hamilton.) (logical) bug fixed (thanks Blaine Bell.) Fixed a bug executing tests in Node2 (thanks Ed Katz.) Fixed a bug with inadvertent changes to current module (thanks John Callahan.) Fixed an (exists) bug and an '|'-constraint bug (thanks Jack Kerkhof).

Version 6.0 (December 7th, 2001):

defclass and definstance now accept defclass names qualified wih module names (thanks Juraj Frivolt.) ppdefrule and friends don't throw if the argument is bad (thanks Mikael Rundqvist.) Release notes and porting info added to docs. Other small patches.

Version 6.0b3 (November 2nd, 2001):

nth$ returns nil instead of throwing exceptions (thanks Seung Lee.) Deffunction has public API for fetching arguments and actions (thanks Scott Kaplan, Henrik Eriksson). If pattern-matches during (modify) throws an exception, the modified fact no longer disappears from the fact-list (thanks Mihai Barbuceanu.) If pattern-matches during addDefrule throw, rule is still added to rulebase (thanks Mikael Rundqvist). Fixed rounding bug in format function (thanks Mike Isenberg). Fixed issue with variable renaming when an 'and' CE was nested in a 'not' CE (thanks Drew Van Duren). Can omit the functor 'call' even if the Java object is the return value of a function call. retract, assert, and duplicate will once again accept an integer as their first argument -- the fact-id function is no longer necessary (although it can still be used.) You can put pattern bindings inside of grouping CEs. (undefinstance *) now works as advertised (thanks Jason Smith). Remove "modify-n" rewrite optimization, so pattern binding to (or) CEs works.

Version 6.0b2 (October 3rd, 2001):

Can redefine a defclass with identical definition (thanks Ian de Beer). Single-stepping rule execution was losing activations (thanks Simon Blackwell.) Fix a bug preventing FuzzyJess from working with non-Fuzzy multifields (thanks Bob Orchard.) explode$ can deal with embedded comments (thanks Simon Blackwell.) (logical) can handle nested (not) and (exists).

Version 6.0b1 (September 13th, 2001):

Fixed a bug in compiling rules with function calls inside of (not (and)) constructs (thanks Thomas Gentsch). Broadcast DEFINSTANCE and DEFCLASS events (ibid.) and RUN and HALT events. defmodules, focus stack, etc., implemented. (clear) doesn't delete Java Userfunctions (thanks Glenn Tarbox.) Rename SerializablePropertyDescriptor to SerializablePD so the class name isn't too long for the Mac filesystem (thanks Matt Bishop). (rules), (facts), others, now in Java, not scriptlib. Redefining a deftemplate throws an exception, unless the definitions are identical. Added "duplicate" function. sym-cat and str-cat call toString() on JAVA_OBJECT arguments (thanks Chad Loder.) Many other bug fixes and cleanups.

Version 6.0a8 (July 19th, 2001):

Docstrings for deftemplates properly parsed (thanks Simon Hamilton.) Several features broken in binary-only distribution (view, ppdefrule, etc) are now fixed. JessEvents of type USERFUNCTION_CALLED now include the Userfunction as the user object, not the FunctionHolder (thanks Chad Loder.)

Version 6.0a7 (June 1st, 2001):

New pretty-print architecture which shows how things are actually represented; based on new Visitor interface. ppdefX functions now return, rather than print, their results. show-jess-listeners works again. Simplify how negated patterns are parsed. JAR file not an executable JAR anymore. Fix Thread problem with runUntilHalt, deadlock in propertyChange (thanks Charles May.) deffunctions accepting only a wildcard can now be called with no arguments (thanks Thomas Gentsch.) No more erroneous undefined variable message (Gentsch.) The minus (-) function works if one or more arguments are atoms, just as the other binary math functions do (thanks Pau Ortega.) The first arg to (modify) can be a funcall. (return) from (try) or (catch) block no longer skips second and subsequent expressions in (finally) block (thanks Chad Loder.) Deadlock in one form of Java assert() (Loder.) (or), (and), and (not) CEs can be nested in structures of arbitrary complexity.

Version 6.0a6 (May 3rd, 2001):

There are no more "optional" functions. Fact.getTime() is public (thanks Alan Moore.) Lots of small improvements to manual. Fix a bug in pattern-matching fact-ids (thanks Matthew Johnson.) Slightly different rules for loading batch files from jars; now they must omit the package-dir name. The "logical" CE is supported.

Version 6.0a5 (March 12th, 2001):

Fixed a serious bug in "modify" (thanks Bob Orchard.) Some documentation cleanup. runQuery method returns Iterator.

Version 6.0a4 (March 8th, 2001):

"Watch" listeners were not being cleared. Use a priority queue for agenda -- modest performance improvement for most applications, big improvement for conflict-resolution-limited apps. Strategy interface vastly simplified, rewrote built-in strategies. FactList.ppFacts(), and hence (save-facts), rewritten to write directly to a Writer, greatly decreasing memory usage. Quit button removed from Console (thanks Andrew Marshall.) Calling a native function that returns a wrapper type (Integer, Float, etc) now gives an JAVA_OBJECT holding the wrapper object, instead of unwrapping the data. Incompatible changes to improve FuzzyJess performance and fix possible bug; a new version of FuzzyJess will be required to work with Jess 6.0a4. Fact-duplication concept

eliminated -- fact duplication is never allowed. Fact list stored on a HashMap, not a Vector; Now finding is fast, enumerating is slow. For the time being, Jess 6.0XX now requires Java 2.

Version 6.0a3 (January 25th, 2001):

Bload and bsave now work much better, using real, complete serialization. Serializing/deserializing Rete saves and restores everything except the I/O router tables. Rete loads scriptlib itself, so you don't need to do it yourself anymore. Lots of refactoring -- Rete now exports more useful functions, but delegates their implementations; for example, there are public defclass(), definstance(), and undefinstance() methods. A small bug in Tokenizer fixed (thanks Norman Ghyra.) Methods added to support Thomas Barnekow's JessX Jess extensions.

Version 6.0a2 (July 20th, 2000):

Defqueries can now backwards-chain to get results. run-until-halt now clears halt flag first. finally in (try) (thanks Thomas Barnekow). Many typos in manual fixed (thanks Michael Futtersack.) (or) and (and) CEs, and general nesting of CEs now operational (thanks to Mariusz Nowostawski and Jack Kerkhof for test cases,) although code still needs refactoring and (and) and (or) won't work correctly inside of (not). waitForActivations() could block even if activations were available; now fixed (thanks Thomas Barnekow). Added some "LISP compatibility functions:" progn, apply. Fixed off-by-one error in line-numbers in error reports. Vastly improved efficiency of multifield matching (thanks to Sebastian Varges for noticing a problem with the old implementation.) Deactivate activations earlier in the rule-firing process, to remove a bit of redundant processing (thanks David Bruce.).

Version 6.0a1 (April 25th, 2000):

Fact-id's are now simply references to Fact objects. Use the supplied fact-id function to create a fact-id value from an integer. Many memory-usage reductions; no more extra Vectors in Rete network classes. Several code-shrinking refactorizations. Rete class no longer delivers events to listeners registered by a handler for the same event (thanks Alex Karasulu.) Fact-ids no longer change on modify. (batch) now works on a .clp files in a .jar file from an applet (thanks Javier Torres).

Version 5.2 (June 30th, 2000):

Bug fix release. waitForActivations() could block even if activations were available; now fixed (thanks Thomas Barnekow).

Version 5.1 (April 24th, 2000):

Bug fix release. Fixed two bugs in backwards chaining (thanks Ashraf Afifi.) Also, a regression repaired: pattern bindings couldn't be matched as they could in Jess 4, and now they can again.

Version 5.0 (January 28th, 2000):

Added new function cross-reference section to the manual (thanks to Roger Firestone for the idea.) parseDeftemplate now handles comments in child templates (thanks Richard Long.) TextAreaWriter (and hence Console and ConsoleApplet) prunes the oldest characters from its TextArea to prevent the Win32 freeze problem.

Version 5.0b4 (January 5th, 2000):

Fixed broken fact-slot-value function. Some support for new RU.LONG type. Added (run-until-halt), runUntilHalt(), activationSemaphore, etc. (unique) CE no longer prevents partial matches from being retracted, only from being propagated (thanks Vicken Kasparian.) Nested nots and (exists) CE added. Unmatched variables can appear in function calls within defqueries (thanks Michal Fadljevic). Bsave/bload now works with defclasses (thanks Russ Milliken.) ReflectFunctions cache result of BeanInfo.getPropertyDescriptors().

Version 5.0b3 (November 30th, 1999):

Support for reworked version of Bob Orchard's fuzzy logic extensions. When multiple activations of one rule were due to different multifield matches of the same set of facts, retracting one fact would cause only one activation to be removed. This is now fixed (thanks, Al Davis.) Share function-call-containing join nodes; thanks George Rudolph for the reminder. Fixed UnWatch class for buggy JDK 1.x compilers. Can match directly on defglobals now. ReflectFunctions cache result of Class.getMethods().

Version 5.0b2 (November 8th, 1999):

Several bugs in run-query and parseDefquery fixed (thanks Alex Karasulu). (watch) handlers flush stdout (thanks David Young.) Fix bare return-value constraints in multislots (thanks Dave Barnett.) Problem with returning parameters from deffunctions, and missing resolveValue calls in BagFunctions fixed (thanks Bruce Douglas.) assert() and retract() call processPendingFacts() directly (thanks Thomas Barnekow.) Multislot matching "optimization" fixed (thanks Robert Gaimari.) Deffacts allows global variables in fact slots (thanks Michael Coen.) Try harder to coerce all multifield entries to same type when converting to Java array (thanks Ralph Grove.) Fix a synchronization problem (thanks Javier Maria Torres Ramon.)

Version 5.0b1 (September 21st, 1999):

jess.Console works again (thanks Lakshmi Vempati). Compiles with buggy JDK 1.1.x compiler. bload/bsave preserves listeners (i.e., deffacts work). Did away with Successor class (simplified Rete network, reduced memory usage.) Fixed bug where redefined rules with shared nodes wouldn't reset. Specialized Value subclasses now explicitly override numericValue(). Yet another pesky "not" bug (thanks Vadim Zaliva.) All facts/definstances inherit, ultimately, from "__fact". In Rete network, callNodeRight() takes Fact, not token; right memories store Facts, too (fewer allocations, less memory!) Return-value constraint was erroneously requiring bound variable; thanks Dave Kirby. Miroslav Madecki reported a "ghost fact" bug; fixing it improved Jess's "manners" benchmark performace by another 25% (!) "system" returns a Process object (thanks Alan Moore). (open) uses a 10-character buffer - large file I/O speedup (thanks Norman Gyhra).

Version 5.0a6 (July 8th, 1999):

insert$, replace$ behave as documented w.r.t argument types (thanks Emmanuel Pierre). Typo: agenda wasn't being cleared on reset (thanks Bob Orchard.) Removed some redundant checks in network (more speedups!) Final multivar argument to deffunction now properly handles passed-in multifields; thanks Ning Zhong. Added "import". Fixed bug in using variable as salience value; thanks David Young. Bug in special code for calling public methods of package-private classes; thanks Cheruku Srini. Bug in member$ fixed (thanks David Young again.) Bug in retracting facts which are multifield pattern-matched on rule LHS fixed (thanks Yang Xiao). All classes serializable again (thanks Michael Friedrich). Several small optimizations related to fact-duplication (thanks Osvaldo Pinali Doederlein.) JessEvents working again, with event mask. Bitwise arithmetic functions added. Fixed some more multifield matching strangeness (thanks Benjamin Good). Added "|" (or) connective constraint.

Version 5.0a5 (May 20th, 1999):

Drastic changes to the way Funcalls are executed (much simplification and speedup!) Fixed "corrupted negcnt" when retracting definstance facts (thanks Abel Martinez.) Much improved agenda management, including time tags. Smart indexing based on tested values. New heuristics for eliminating redundant tests. (store foo nil) now removes value of foo (thanks Kathy Lee Simunich for the suggestion.) (system) command now pipes program output to console (thanks Fang Liu for idea). (undefinstance *) now removes all definstances (thanks Mike Lucero for idea). Multiple defglobal references on rule LHS no longer broken (thanks Jacek Gwizdka for report.) Many upgrades to JessException class

(thanks Kenny Macleod). Some Value constructors no longer have the redundant second argument. get/set-fact-duplication (thanks to Osvaldo Pinali Doederlein for the idea.) Bob Orchard figured out how to get rid of the extra EOFs in jess.ConsolePanel. Console and ConsoleApplet can now read a file and continue, as they used to. Backwards chaining now more general; one goal can trigger multiple rules, both in parallel and in chains. ViewFunctions and ReflectFunctions moved into jess package, many more classes now package-private. jess.reflect package renamed jess.awt; TextReader, TextAreaWriter moved into it. "throw" command added (thanks Eric Eslinger).

Version 5.0a4 (March 18th, 1999):

Call removePropertyChangeListener on undefinstance (thanks to Dan Larner.) Added defadvice. Three buglets, thanks to Bob Orchard: defglobals can now refer to other Defglobals in their definitions; (bind) and (foreach) are now pickier about arguments; undefined variables now universally evaluate as 'nil'. Applet security problems fixed (thanks S.S. Ozsariyildiz). Problem with matching and otherwise dealing with zero-length multislots and array bean properties fixed; null array values are converted into zero-length arrays (thanks Miroslav Madecki.) (call) does much better error reporting. Passing Strings to functions accepting Objects now works (J. P. van Werkhoven.)

Version 5.0a3 (February 12th, 1999):

Token(Token) changes UPDATE to ADD. Hashcode used in TokenTree modifiable. (clear) now fully clears deffacts. Many general speedups. TokenTree hash factor can now be set globally and per-rule. Functional Accelerator class exists for math functions. (modify) funcalls print out correctly. get-var removed (no longer needed.) Nonsychronization error in jess.Retract.call() led to Corrupted Negcnt; fixed (reported by Dan Larner). Other corrupted negcnt error due to duplicated tokens fixed. definstances can be static or dynamic. jess.Pattern is public; several methods added or made public to support rule editors. Defrule.toString works for all cases. Console class now uses Main to do business. .jessrc file now named scriptlib.clp and found using Class.getResource(). (batch) can use getResource() to find scripts, so they can live in .jar files. Batch now an intrinsic. Rewrote some multifield functions to better handle JAVA_OBJECT objects; added hashCode() to jess.Value. (facts) and (rules) now in scriptlib, not Java. Much improved public interface for Deftemplate, Fact classes.

Version 5.0a2 (December 15th, 1998):

m_sortcode in Token no longer a blank final to work around JDK 1.1.6/7a javac bug. Makefile no longer mentions unbundled files. Backward chaining works again (was broken in 5.0a1)

Version 5.0a1 (December 9th, 1998):

All 16-bit (Reader/Writer) text I/O - some user code changes required. Serialization (crude version). Inheritance for deftemplates, defclasses. jess.Main reads $HOME/.jessrc at startup if it exists. No more ReteDisplay, Resetable, or Clearable interfaces: instead, use jess.JessListener and jess.JessEvent. (view) uses JessListener instead of Observer/Observable; rewritten to provide movable nodes, per-node debug code. jess.reflect.JessListener class renamed to jess.reflect.JessAWTListener. Bean-itized many function names (name() -> getName()), breaking lots of user code, I'm afraid. (run) returns number of rules fired. Ordered facts now stored as unordered with one multislot named __data. Backwards chaining added. Bload and Bsave added.

Version 4.5 (April 15th, 1999):

Fixed "corrupted negcnt" when retracting definstance facts (thanks Abel Martinez.)

Version 4.4 (March 18th, 1999):

Token(Token) changes UPDATE to ADD. (clear) not truly clearing deffacts (thanks to Rob Jefson). Passing Strings to Java methods expecting Object now works. Call removePropertyChangeListener in undefinstance.

Version 4.3 (December 1st, 1998):

Fixed redundant default-value processing, which was leading to odd problems with definstances with null slot values (thanks to S.S. Ozsariyildiz). Removed intern()s from Tokenizer (faster compilation). Fixed NIL/nil ambiguity in ReflectFunctions (thanks Andreas Rasmussen.)

Version 4.2 (November 12th, 1998):

Fixed 'Corrupted negcnt' bug (thanks to Todd Bowers). (if ... then) function now throws an exception if atom 'then' is missing. Version string in 4.1 final was inadvertently left at 4.1b6. Added section to README explaining rule LHS semantics a bit better. Rete.findFactByID() is now public. Fix for very tricky 'phantom fact' problem reported by Steve Bucuvalas. Java method calls on Jess Strings now work for all Strings, not just alphanumeric ones. "animals" example modified to work with transitional gensym implementation.

Version 4.1 (September 15th, 1998):

Some minor bug fixes; code to allow you to leave off the '$' on a multivar after its first use, as in CLIPS.

Version 4.1b6:

Allow named variables in (not) CEs as long as they're not used in subsequent CEs. Fix a bug that was causing (return) to not work if inside a (foreach) inside a deffunction. Recursive deffunctions now work again. Jess works around a bug in some versions of Java that was preventing the atom '-' from parsing. Rete.listDefglobals() no longer lists duplicates of redefined defglobals (Karl Mueller found this one.) ReteDisplay.fireRule() is now called as appropriate. Accessing pattern-binding variables on rule LHSs works again (Karl again.) (reset) wasn't clearing all activations (thanks Al Davis); fixed. Funcall.toString() puts parens around the ValueVector version.

Version 4.1b5:

Just remove some debug code and extra files inadvertently shipped with 4.1b4.

Version 4.1b4:

addUserfunction, addDeffunction, etc collapsed into one addUserfunction routine in Rete class; same with findUserfunction. RU.getAtom() and RU.putAtom are gone! Userfunction.name() now returns String. ControlStructure interface used to clean up handling of such things. ReteCompiler uses Hashtables of Bindings instead of int[][] for vartables. Added default-dynamic deftemplate slot qualifier. Added set-/get-reset-globals, and changed the default defglobal reset behaviour. Added dynamic rule salience. Removed arbitrary limit of 32 slots for ordered facts and 32 tests per slot for all facts. "unique" CE (15-30% speed increase for many problems!) Various documentation improvements (many thanks to Win Carus.) Better error reporting (addContext() call in JessException.) Malformed calls to 'eval' or 'build' or 'executeCommand' no longer go into an infinite loop on EOF. Added "store" and "fetch". Added "external-addressp". Rearranged Test1, Test2 classes into an inheritance hierarchy with a virtual doTest method, allowing for alternate implementations (undocumented java-test functionality included). Value class will do more type conversions automatically. Final multifield argument of a deffunction now acts as a wildcard, as in CLIPS (thanks David Young.)

Version 4.1b3

Problem with calling public methods of package-private classes from Jess fixed thanks to Lars Rasmusson's explanation. OutOfMemoryError while parsing file containing unbalanced open parens fixed. Line breaks in double-quoted strings no longer need to be (but can be) escaped. Two fixes thanks to Andreas Rasmusson: gensym* returns a

symbol as documented, not a string; and a propertyChangeEvent for a bogus property no longer causes Jess to retract a definstance without updating it. Many of the synchronized methods in the Rete class no longer are synchronized; instead they use either synchronized blocks keyed to affected members or simply depend on the internal synchronization of Hashtables. read and readline explicitly act differently for console-like and file-like streams. ConsoleDisplay gets a Clear Window button.

Version 4.1b2

Bug in character-escape lexing fixed thanks to Josiah Poon. Parser-related bug in explode$ fixed thanks to Andrew X. Comas. eval, build, executeCommand() again properly return the result of last expression, not EOF. min, max take arbitrary # of arguments. implode$ now works; it apparently never really did. printout puts parens around multifields again. str-compare documentation corrected. undefinstance now removes the fact representing an object as well as deactivating matching. Wrote large regression test suite (not included in distribution). Bug in multiple simultaneous Rete.run() calls in separate threads fixed thanks to Andreas Rasmusson. Selectable conflict resolution strategies (only depth and breadth supported now) and user-definable strategies. The try command is added.

Version 4.1b1

Much better lexer (no more StreamTokenizer). Input buffering problems with JDK 1.1.2-1.1.4 fixed. Bug in (test) CE fixed. Can call run on rule RHS. Bug in incremental update fixed. Separate command-line, applet, and GUI console driver classes (Quiz* classes broken up, renamed to Console*). read and readline should work exactly as in CLIPS. Manual describes more about how to write Java main(). Bug in definstance that was preventing use of subclasses of a defclassed class is fixed.

Version 4.0

BeanInfo support. quiz.html embeds only one QuizDisplay applet. Pumps demo works again (sorry). Conflict resolution strategy now should be exactly the same as CLIPS's default.

Version 4.0b4

Extensive manual rewrite, adding lots of Java/Jess interoperation info. Allow standard CLIPS deffunction docstrings. Thanks to Jack Fitch, Dave Carlson and Alex Jacobson, property names for reflected Java Beans now use standard capitalization transform. Better error reporting, especially during parsing and from the command line. set and get renamed to set-member and get-member. set and get are now functions that read and write Bean properties. ppdefrule properly handles quoted strings in function calls. executeCommand and friends reuse a single parser. Thanks to Karl Mueller for Rete.retractString. Taught batch to read applet-based data files. eval now handles non-sexps. Better mechanism for synchronizing streams. QuizDisplay is an applet as well as an application. run accepts an argument, the maximum number of rules to fire. Fixed bug in modify when new slot value was a zero-length multifield. Fixed ReteCompiler bug where MTELN nodes were not consistently generated for zero-length multifield matches. Thanks to Sidney Bailin, fixed problem with accessing defglobals and variables bound to pattern indexes on rule LHSs. Added get-var function. Added undefinstance. modify and retract now handle definstance facts specially. Fixed some doPPPattern bugs (Dave Carlson again!).

Version 4.0b3

Added jess.reflect package containing new, call, set, and get. Added JessListener and its subclasses. added engine. Changed printing of external-addresses to include Java class name. Changed parser to accept variable names as Funcall heads (call is substitued, resulting in a runtime error if call is not installed). and and or functions now accept any values as arguments, not only funcalls. Added foreach control structure.

Command prompt doesn't print NIL return values. Fixed another not bug (thanks to Sidney Bailin). Added matching of Java objects on rule LHSs: definstance, defclass. TokenTree now uses sortcode % 101 as hash key, not the sortcode itself. All global classes moved into jess package. Jess class renamed Main.

Version 4.0b2

Cleaned up router/parser interactions. Jess will now read only one construct on a line of input (just like CLIPS). All Jess output now goes through WSTDOUT router, not through ReteDisplay.stdout(). Fixed bug whereby second and later references to subfields of multifields on the LHS of a rule would resolve to the whole multifield. modify can now properly handle multislots. format handles trailing spaces. Finally, parsing of integers: 2 is an RU.INTEGER, while 2.0 is an RU.FLOAT. Added eval and list-function$.

Version 4.0b1

Code reformat. Major performance enhancements (Value and Funcall recycling; Fastfunction interface; Rete memories are now btrees; RU.CLEAR tokens). test CE. Return-value constraints. ppdefrule thanks to Rajaram Ganeshan. Blank variables in not CEs. system blocks by default. readline fixed. build supported. logic for predicate functions in Rete network now precisely the same as for CLIPS. QuizDisplay demo. while and if accept boolean variables. Implied returns from if and while functions. Added explode$. Added I/O routers: open, close. Added format. Added bag.

Version 3.2

system and integer Userfunction classes renamed (Win95 filename capitalization problem!). Broken delete$, insert$, replace$ fixed. view command added. Big if/then in Funcall class finally removed in favor of separate implementation classes for intrinsics, leading to a modest speed increase. Documentation vastly expanded! Added catch for ArrayOutOfBoundsException in command-line interface; no more crash on wrong number of arguments. Broken evenp, oddp fixed. str-cat, sym-cat made more general. Broken sub-string fixed. Big switch in Node1 class replaced by separate classes, leading to a very modest speed increase.

Version 3.1

Added the assert-string and batch commands. Two bug fixes in multislot code (thanks to Nancy Flaherty). Added undefrule and the ability to redefine rules. Added the system function, although it doesn't work very well under Java. Public function engine() in jess.Context class allows you to do fancier things in Userfunctions. Added the non-standard load-package and load-function functions. Many new contributed functions packaged with Jess for doing math, handling multifields, and other neat stuff thanks to Win Carus. Added time (1 second resolution).

Version 3.0

A few code changes to accomodate Microsoft's Java compiler; Jess now compiles unchanged with JVC thanks to Mike Finnegan. Added member$ multifield function. Added clear intrinsic thanks to Karl Mueller. Introduced a new way of handling not patterns which I think finally guarantees there are no more not-related bugs remaining! load-facts, which has been non-functional throughout the beta period, is working again. Documentation now explains unzipping and compiling a little better. Modified the way fact-id's are handled so that you can write (retract 3) to retract fact #3.

Version 3.0b2

Lots of bug reports and improvement suggestions from the field - thanks folks! All the reported bugs in the multifield implementation, and some residual odd behavior in the not CE, have been fixed. The exit command has been added. A command prompt has been added. The # character can now be used in symbols. The access levels on some methods in the Rete class have been opened up; Rete is no longer final. nth$ is now 1-based, as it is in CLIPS. The if and while constructs now fire on not FALSE instead of

TRUE. The str-index function has been fixed and added. Probably a few more things I'm forgetting here. Thanks for the input. Particular thanks to Nancy Flaherty, Joszef Toth, Karl Mueller, Duane Steward, and Michelle Dunn for reporting bugs fixed in this version; sorry if I left anyone out.

Version 3.0b1

First public release of Jess 3.0.

Version 3.0a3

UserPackage interface. Lots of new example UserFunctions for multifields, string, and predicates.

Version 3.0a2

Multislots! Also important bug fix: under certain circumstances, the Rete network compilation could fail 1) if (not()) CEs occurred on the LHS of a rule, 2) new variables were introduced in that rule's patterns listed after the (not()) CEs, and 3) these latter variables were tested (i.e., in a predicate constraint) on the LHS of the rule.

Version 3.0a1

Incremental reset. Watch activations. gc() in LostDisplay, NullDisplay. Multifields! All the Rete engine classes are now in a package named jess. Many classes and methods that should not be manipulated by clients are now package-private.

Version 2.2.1

Ken Bertapelle found another bug, which has been squashed, in the pattern network.

Version 2.2

Jess 2.2 adds a few new function calls (load-facts, save-facts) and fixes a serious bug (thanks to Ken Bertapelle for pointing it out!) which caused Jess to crash when predicate constraints were used in a certain way. Another bug fix corrected the fact that retract only retracted the first of a list of facts. Jess used to give a truly inscrutable error message if a variable was first used in a not CE (a syntax error); the current error message is much easier to understand. I also clarified a few points in the documentation.

Version 2.1

Jess 2.1 is *much* faster than version 2.0. The Monkey example runs in about half the time as under Jess 2.0, and for some inputs, the speed has increased by an order of magnitude! This is probably the last big speed increase I'll get. For Java/Rete weenies, this speed increase came from banishing the use of java.lang.Vector in Tokens and in two-input node memories. Jess is now within a believable interpreted Java/C++ speed ratio range of about 30:1. Jess 2.1 now includes rule salience. It also implements a few additional intrinsic functions: gensym*, mod, readline. Jess 2.1 fixes a bug in the way predicate constraints were parsed under some conditions by Jess 2.0. The parser now reports line numbers when it encounters an error.

Version 2.0

Jess 2.0 is intrinsically about 30% faster than version 1.0. The internal data structures changed quite a bit. The Rete network now shares nodes in the Join network instead of just in the pattern network. The current data structures should allow for continued improvement.

Index