Jess Tutorial

Maarten Menken <mrmenken@cs.vu.nl>
Vrije Universiteit, Amsterdam, The Netherlands

December 24, 2002

Contents

1 Facts **7**
1.1 Introduction . . . 7
1.2 The Beginning And The End . . . 7
1.3 Making A List . . . 8
1.4 And Checking It Twice . . . 8
1.5 Clearing Up The Facts . . . 9
1.6 Fields . . . 10
1.7 Retract That Fact . . . 11

2 Rules **13**
2.1 Making Good Rules . . . 13
2.2 The Agenda . . . 15
2.3 Write To Me . . . 16
2.4 Being Efficient . . . 17
2.5 Other Features . . . 17

3 Adding Details **19**
3.1 Stop And Go . . . 19
3.2 Take A Walk . . . 19
3.3 A Question Of Strategy . . . 20
3.4 Deffacts . . . 21
3.5 Debugging . . . 22

4 Variables **23**
4.1 Let's Get Variable . . . 23
4.2 Be Assertive . . . 24
4.3 What The Duck Said . . . 24
4.4 The Happy Bachelor . . . 25
4.5 It's not Important . . . 27
4.6 Going Wild . . . 28
4.7 Bound Variables . . . 30
4.8 The Lucky Duck . . . 30

5 Deftemplates **33**
5.1 Mr. Wonderful . . . 33
5.2 Bye-Bye . . . 35
5.3 Ain't No Strings On Me . . . 36
5.4 What's In A Name . . . 37

6 Functions **39**
6.1 Doing Your Own Thing 39

7 Modules **41**
7.1 Introduction 41
7.2 Defining Constructs In Modules 42
7.3 Modules, Scope And Name Resolution 43
7.4 Module Focus And Execution Control 44
7.4.1 The Auto-Focus Declaration 45
7.4.2 Returning From A Rule Right-Hand-Side 45

8 Fuzzy Logic **47**
8.1 Introduction 47
8.2 Fuzziness 47
8.3 Example 48
8.4 Example Results 54

Introduction

This article is an introductory tutorial on the basic features of Jess. It is *not* intended to be a comprehensive discussion of the entire tool. Prerequisites are that you have a basic knowledge of expert systems and programming in a high-level language such as Java or C.

It should be emphasized that this text is merely a compilation of the book and web pages listed in the bibliography. Most of the text is copied, adapting the examples and source code to Jess.

What Is Jess?

Jess is an acronym for *J*ava *E*xpert *S*ystem *S*hell. It is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, Canada. Jess was originally inspired by the CLIPS expert system shell, but has grown into a complete, distinct Java-influenced environment of its own. Using Jess, you can build Java applets and applications that have the capacity to "reason" using knowledge you supply in the form of declarative rules. Like CLIPS, Jess has a Lisp-like syntax, which stands either for "List Processing" or "Lots of Irritating, Superfluous Parentheses," depending on who you ask.

Jess is a tool for building a type of intelligent software called expert systems. An expert system is a set of rules that can be repeatedly applied to a collection of facts about the world. It is specifically intended to model human expertise or knowledge. Rules that apply are fired, or executed. Jess uses a special algorithm called Rete to match the rules to the facts.

There are three ways to represent knowledge in Jess:

- *Rules*, which are primarily intended for heuristic knowledge based on experience.
- *Functions*, which are primarily intended for procedural knowledge.
- *Object-oriented programming*, also primarily intended for procedural knowledge. The five generally accepted features of object-oriented programming are supported: classes, message-handlers, abstraction, encapsulation, inheritance, polymorphism. Rules may pattern match on objects and facts.

You can develop software using only rules, only objects, or a mixture of objects and rules.

Chapter 1

Facts

This chapter introduces the basic concepts of an expert system. You'll see how to insert and remove facts in Jess.

1.1 Introduction

Jess is called an expert system *tool* because it is a complete environment for developing expert systems which includes features such as an integrated editor and a debugging tool. The word *shell* is reserved for that portion of Jess which performs *inferences* or reasoning. The Jess shell provides the basic elements of an expert system:

- *fact-list* and *instance-list*: global memory for data
- *knowledge-base*: contains all the rules, the *rule-base*
- *inference engine*: controls overall execution of rules

A program written in Jess may consist of *rules*, *facts* and *objects*. The inference engine decides which rules should be executed and when. A rule-based expert system written in Jess is a data-driven program where the facts, and objects if desired, are the data that stimulate execution via the inference engine.

This is one example of how Jess differs from procedural languages such as Java and C. In procedural languages, execution can proceed without data. That is, the statements are sufficient in those languages to cause execution. For example, a statement such as `PRINT 2 + 2` could be immediately executed in BASIC. This is a complete statement that does not require any additional data to cause its execution. However, in Jess, data are required to cause the execution of rules.

1.2 The Beginning And The End

To begin Jess, just enter the appropriate run command for your system. You should see the Jess prompt appear as follows:

```
Jess, the Java Expert System Shell
Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
Jess Version 6.0 12/7/2001

Jess>
```

At this point, you can start entering commands directly into Jess. The mode in which you are entering direct commands is called the *top-level*.

The normal mode of leaving Jess is with the `exit` command. Just type

```
(exit)
```

in response to the Jess prompt and press the carriage return key.

1.3 Making A List

As with other programming languages, Jess recognizes certain keywords. For example, if you want to put data in the fact-list, you can use the `assert` command.

As an example of `assert`, enter the following right after the Jess prompt as shown:

```
Jess> (assert (duck))
```

Here the `assert` command takes `(duck)` as its argument. You will see the response

```
<Fact-0>
```

which indicates Jess has stored the `(duck)` fact in the fact-list and given it the identifier 0. The angle-brackets are used as a delimiter in Jess to surround the name of an item. Jess will automatically name facts using a sequentially increasing number and list the highest fact-index when one or more facts is asserted. Notice that the `assert` command and its `(duck)` argument are surrounded by parentheses. Like many other expert system languages, Jess has a LISP-like syntax which uses parentheses as delimiters. Although Jess is not written in LISP, the style of LISP has influenced the development of Jess.

1.4 And Checking It Twice

Suppose you want to see what's in the fact-list. The keyboard command to see facts is `facts`. Enter `(facts)` in response to the Jess prompt and Jess will respond with a list of facts in the fact-list. Be sure to put parentheses around the command or Jess will not accept it. The result of the `facts` command in this example should be

```
Jess> (facts)
f-0   (MAIN::duck)
For a total of 1 facts.
Jess>
```

The term `f-0` is the *fact identifier* assigned to the fact by Jess. Every fact inserted into the fact-list is assigned a unique fact identifier starting with the letter `f` and followed by an integer called the *fact-index*. On starting up Jess, and after certain commands such as `clear` and `reset` (to be discussed in more detail later), the fact-index will be set to zero, and then incremented by one as each new fact is asserted. The `reset` command will also insert a fact `(initial-fact)` as `f-0`. This fact is often used for convenience to initially activate rules. Shown following is what happens when a `reset` is done first.

```
Jess> (reset)
TRUE
Jess> (facts)
f-0   (MAIN::initial-fact)
For a total of 1 facts.
Jess> (assert (duck))
<Fact-1>
Jess> (facts)
f-0   (MAIN::initial-fact)
f-1   (MAIN::duck)
For a total of 2 facts.
Jess>
```

Notice that the fact-index is `<Fact-1>` after the `(duck)` fact is asserted because there are now two facts in working memory and the duck has index 1.

Facts may be removed or *retracted*. When a fact is retracted, the other facts do not have their indices changed, and so there may be “missing” fact-indices.

Jess is said to be case-sensitive because it distinguishes between uppercase and lowercase letters. For example, assert the facts `(duck)` and `(Duck)` and then issue a `facts` command. You’ll see that Jess allows you to assert `(duck)` and `(Duck)` as different facts because Jess is case-sensitive.

1.5 Clearing Up The Facts

The `clear` command removes all facts from memory, as shown by the following.

```
Jess> (facts)
f-0   (MAIN::initial-fact)
f-1   (MAIN::duck)
For a total of 2 facts.
Jess> (clear)
TRUE
Jess> (facts)
For a total of 0 facts.
Jess>
```

The `clear` command essentially restores Jess to its original startup state. It clears the memory of Jess and resets the fact-identifier to zero. Besides removing all the facts, `clear` also removes all the rules, as you’ll see in the next chapter.

1.6 Fields

A fact such as `(duck)` is said to consist of a single *field*. A field is a placeholder (named or unnamed) that may have a value associated with it. Named placeholders are only used with `deftemplates`, described in more detail in Chapter 5. The `(duck)` fact has a single, unnamed placeholder for the value duck. This is an example of a *single-field* fact.

The *order* of unnamed fields is significant. For example, if a fact was defined

```
(Brian duck)
```

and interpreted by a rule as the hunter Brian shot a duck, then the fact

```
(duck Brian)
```

would mean that the hunter duck shot a Brian. In contrast, the order of named fields is not significant, as you'll see later with `deftemplate`.

Actually, it is good software engineering to start the fact with a relation that describes the fields. A better fact would be

```
(hunter-game Brian duck)
```

to imply that the first field is the hunter and the second field is the game.

A few definitions are now necessary. A *list* is a group of items with no implied order. Saying that a list is *ordered* means that the position in the list is significant. A *multifield* is a sequence of fields, each of which may have a value. The examples of `(Brian duck)`, `(duck Brian)` and `(hunter-game Brian duck)` are multifield facts.

There are a number of different types of fields available: `INTEGER`, `LONG`, `FLOAT`, `ATOM`, `STRING`, `LIST` and `EXTERNAL_ADDRESS`. The type of each field is determined by the type of value stored in the field. In an unnamed field, the type is determined implicitly by what type you put in the field. In deftemplates, you can explicitly declare the type of value that a field can contain. The use of explicit types enforces the concepts of software engineering, which is a discipline of programming to produce quality software.

A fact consists of one or more fields enclosed in matching left and right parentheses. A fact may be ordered or unordered. All the examples you've seen so far are ordered facts because the order of fields makes a difference. For example, notice that Jess considers these as separate facts although the same values `1`, `2`, and `3` are used in each.

```
f-0 (MAIN::coordinates 1 2 3)
f-1 (MAIN::coordinates 1 3 2)
```

Ordered facts *must* use field position to define data. As an example, the ordered fact `(Brian duck)` has two fields and so does `(duck Brian)`. However, these are considered as two separate facts by Jess because the order of field values is different. In contrast, the fact `(Brian-duck)` has only one field because of the `-` concatenating the two values. `deftemplate` facts, described in more detail later, are unordered because they use named fields to define data. Multiple fields normally are separated by white space consisting of one or more spaces, tabs, carriage returns, or linefeeds.

The following example is a more realistic case in which carriage returns are used to improve the readability of a list. To see this, assert the following fact where carriage returns and spaces are used to put fields at appropriate places on different lines.

```
Jess> (clear)
TRUE
Jess> (assert (grocery-list
    ice-cream
    cookies
    candy
    fudge-sauce))
<Fact-0>
Jess> (facts)
f-0   (MAIN::grocery-list ice-cream cookies candy fudge-sauce)
For a total of 1 facts.
Jess>
```

As you can see, Jess replaced the carriage returns and tabs with single spaces.

It is good rule-based programming style to use the first field of a fact to describe the relationship of the following fields. When used this way, the first field is called a relation. The remaining fields of the fact are used for specific values. An example is `(grocery-list ice-cream cookies candy fudge-sauce)`.

1.7 Retract That Fact

Now that you know how to put facts into the fact-list, it's time to learn how to remove them. Removing facts from the fact-list is called *retraction* and is done with the `retract` command. To retract a fact, you must specify the fact-index of the fact as the argument of retract. For example, set up your fact-list as follows.

```
Jess> (assert (animal-is duck))
<Fact-0>
Jess> (assert (animal-sound quack))
<Fact-1>
Jess> (assert (The duck says "Quack."))
<Fact-2>
Jess> (facts)
f-0   (MAIN::animal-is duck)
f-1   (MAIN::animal-sound quack)
f-2   (MAIN::The duck says "Quack.")
For a total of 3 facts.
Jess>
```

To retract a fact, you must specify the fact-index. To remove the last fact with index `f-2`, enter the retract command and then check your facts as follows.

```
Jess> (retract 2)
TRUE
```

```
Jess> (facts)
f-0   (MAIN::animal-is duck)
f-1   (MAIN::animal-sound quack)
For a total of 2 facts.
Jess>
```

Chapter 2

Rules

In the previous chapter, you learned about facts. Now you'll see how the rules of an expert system utilize facts in making a program execute.

2.1 Making Good Rules

To accomplish useful work, an expert system must have rules as well as facts. Since you've seen how facts are asserted and retracted, it's time to see how rules work. A rule is similar to an IF-THEN statement in a procedural language like Java or C. An IF-THEN rule can be expressed in a mixture of natural language and computer language as follows:

```
IF   certain conditions are true
THEN execute the following actions
```

The pseudocode for a rule about duck sounds might be

```
IF   the animal is a duck
THEN the sound made is quack
```

The following is a fact, and a rule named `duck-sound` which is the pseudocode above expressed in Jess syntax. The name of the rule follows immediately after the keyword `defrule`.

```
Jess> (clear)
TRUE
Jess> (assert (animal-is duck))
<Fact-0>
Jess> (defrule duck-sound
    (animal-is duck)
  =>
    (assert (sound-is quack)))
TRUE
Jess>
```

If you type in the rule correctly as shown, you should see the Jess prompt reappear. Otherwise, you'll see an error message. If you get an error message,

it is likely that you misspelled a keyword or left out a parenthesis. Remember, the number of left and right parentheses always must match in a statement.

The same rule is shown following with comments added to match the parts of the rule. Also shown is the optional *rule header* comment in quotes, `"Here comes the quack"`. There can be only one rule-header comment and it must be placed after the rule name and before the first *pattern*. Jess tries to match the pattern of the rule against facts. White space consisting of spaces, tabs, and carriage returns may be used to separate the elements of a rule to improve readability. Other comments begin with a semicolon and continue until the carriage return key is pressed to terminate a line. Comments are ignored by Jess.

```
(defrule duck-sound
    "Here comes the quack"    ; rule header

    (animal-is duck)          ; pattern
  =>                          ; THEN arrow
    (assert (sound-is quack)) ; action
)
```

Only one rule name can exist at one time in Jess. Entering the same rule name, in this case `duck-sound`, will replace any existing rule with that name. That is, while there can be many rules in Jess, there can be only one rule which is named `duck-sound`. This is analogous to other programming languages in which only one procedure name can be used to uniquely identify a procedure.

The general syntax of a rule is shown following.

```
(defrule rule-name
    "optional comment"

    (pattern-1)         ; left-hand side (LHS) of the rule
    (pattern-2)         ; consisting of elements before the "=>"

    (pattern-n)
  =>
    (action-1)          ; right-hand side (RHS) of the rule
    (action-2)          ; consisting of elements after the "=>"

    (action-m)
)                       ; the last ")" balances the opening "(" to
                        ; the left of "defrule". Be sure all your
                        ; parentheses balance or you will get
                        ; error messages.
```

The entire rule must be surrounded by parentheses. Each of the rule patterns and actions must be surrounded by parentheses. An action is actually a function which typically has no return value, but performs some useful action, such as an `assert` or `retract`. For example, an action might be `(assert (duck))`. Here the function name is `assert` and its argument is `duck`. Notice that we don't want any return value such as a number. Instead, we want the fact `(duck)` to be asserted. A function in Jess is a piece of executable code identified by a

specific name, which returns a useful value or performs a useful side-effect, such as `printout`.

A rule often has multiple patterns and actions. Zero or more patterns may be written after the rule header. Each pattern consists of one or more fields. In the `duck` rule, the pattern is `(animal-is duck)`, where the fields are `animal-is` and `duck`. Jess attempts to match the patterns of rules against facts in the fact-list. If all the patterns of a rule match facts, the rule is *activated* and put on the *agenda*. The agenda is a collection of activations which are those rules which match pattern entities. Zero or more activations may be on the agenda.

The symbol `=>` that follows the patterns in a rule is called an arrow. The arrow represents the beginning of the THEN part of an IF-THEN rule (and may be read as "implies").

The last part of a rule is the list of zero or more actions that will be executed when the rule fires. In our example, the one action is to assert the fact `(sound-is quack)`. The term fires means that Jess has selected a certain rule for execution from the agenda.

A program will cease execution when no activations are on the agenda. When multiple activations are on the agenda, Jess automatically determines which activation is appropriate to fire. Jess orders the activations on the agenda in terms of increasing priority or *salience*.

2.2 The Agenda

Jess always executes the actions on the right-hand side of the highest priority rule on the agenda. This rule is then removed from the agenda and the actions of the new highest salience rule is executed. This process continues until there are no more activations or a command to stop is encountered.

You can check what's on the agenda with the `agenda` command. For example,

```
Jess> (agenda)
[Activation: MAIN::duck  f-0 ; time=2 ; salience=0]
For a total of 1 activations.
Jess>
```

`f-0` is the fact-identifier of the fact, `(animal-is duck)`, which matches the activation. If the salience of a rule is not declared explicitly, Jess assigns it the default value of zero.

If there is only one rule on the agenda, that rule will fire. Since the left-hand side pattern of the `duck-sound` rule is

```
(animal-is duck)
```

this pattern will be satisfied by the fact `(animal-is duck)` and so the `duck-sound` rule should fire.

Each field of the pattern is said to be a *literal constraint*. The term literal means having a constant value, as opposed to a variable whose value is expected to change. In this case, the literals are `animal-is` and `duck`.

To make a program run, just enter the `run` command. Type `(run)` and press the carriage return key. Then do a `(facts)` to check that the fact was asserted by the rule.

```
Jess> (run)
1
Jess> (facts)
f-0   (MAIN::animal-is duck)
f-1   (MAIN::sound-is quack)
For a total of 2 facts.
Jess>
```

An interesting question may occur to you at this time. What if you `(run)` again? There is a rule and a fact which satisfies the rule, so the rule should fire. However, if you try this and `(run)` again, you'll see that the rule won't fire. This may be somewhat frustrating. A rule is activated if its patterns are matched by

- a brand new pattern entity that did not exist before or,
- a pattern entity that did exist before but was retracted and reasserted, i.e., a "clone" of the old pattern entity, and thus now a new pattern entity.

The rule, and indices of the matching patterns, is the activation. If either the rule or the pattern entity, or both change, the activation is removed. An activation may also be removed by a command or an action of another rule that fired before and removed the conditions necessary for activation.

The inference engine sorts the activations according to their salience. This sorting process is called *conflict resolution* because it eliminates the conflict of deciding which rule should fire next. Jess executes the right-hand side of the rule with the highest salience on the agenda, and removes the activation.

2.3 Write To Me

Besides asserting facts in the right-hand side of rules, you also can print out information using the `printout` function. Jess also has a carriage return/linefeed keyword called `crlf` which is very useful in improving the appearance of output by formatting it on different lines. As an example,

```
Jess> (defrule duck-print
    (animal-is duck)
  =>
    (printout t "Quack!" crlf) ; be sure to type in the "t"
)
TRUE
Jess> (run)
Quack!
1
Jess>
```

The output is the text within the double quotes. Be sure to type the letter `t` following the `printout` command. This tells Jess to send the output to the standard output device of your computer. Generally, the standard output device is your terminal (hence the letter `t` after `printout`.) However, this may be redefined so that the standard output device is some other device, such as a modem or disk.

2.4 Being Efficient

Jess is a rule-based language that uses a very efficient pattern-matching algorithm called the *Rete Algorithm*, devised by Charles Forgy of Carnegie-Mellon University for his OPS shell. The term *rete* is Latin for net, and describes the software architecture of the pattern-matching process.

It is very difficult to give precise rules that will always improve the efficiency of a program running under the Rete Algorithm. However, the following should be taken as general guidelines that may help:

1. Put the most specific patterns in a rule first. Patterns with unbound variables and wildcards should be lower down in the list of rule patterns. A control fact should be put first in the patterns.

2. Patterns with fewer matching facts should go first to minimize partial matches.

3. Patterns that are often retracted and asserted, *volatile patterns*, should be put last in the list of patterns.

As you can see, these guidelines are potentially contradictory. A non-specific pattern may have few matches (see guidelines 1 and 2). Where should it go? The overall guideline is to minimize changes of the partial matches from one cycle of the inference engine to the next. This may require much effort by the programmer in watching partial matches. An alternative solution is simply to buy a faster computer, or an accelerator board. This is becoming more attractive since the price of hardware always goes down while the price of human labor always goes up. Because Jess is designed for portability, any code developed on one machine should work on another.

2.5 Other Features

The `declare (salience)` command provides explicit control over which rules will be put on the agenda. You must be careful in using this feature too freely lest your program become too controlled.

The `batch` command allows you to execute commands from a file as if they were typed in at the shell. By convention, the extension `clp` is used for files containing Jess code. For example, if you want to load the code from the file `hello.clp`, use the following command.

```
Jess> (batch hello.clp)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Hello, world!
1
Jess>
```

Chapter 3

Adding Details

In the first two chapters, you learned the fundamentals of Jess. Now you will see how to build on that foundation to create more powerful programs.

3.1 Stop And Go

Until now, you've only seen the simplest type of program consisting of just one rule. However, expert systems consisting of only one rule are not very useful. Practical expert systems may consist of hundreds or thousands of rules. Let's now take a look at an application requiring multiple rules.

Suppose you wanted to write an expert system to determine how a mobile robot should respond to a traffic light. It is best to write this type of problem using multiple rules. For example, the rules for the red and green light situations can be written as follows.

```
(defrule red-light
    (light red)
  =>
    (printout t "Stop" crlf)
)

(defrule green-light
    (light green)
  =>
    (printout t "Go" crlf)
)
```

After the rules have been entered into Jess, assert a fact `(light red)` and run. You'll see `Stop` printed. Now assert a `(light green)` fact and run. You should see `Go` printed.

3.2 Take A Walk

If you think about it, other possibilities beside the simple red, green, and yellow cases exist. Some traffic lights also have a green arrow for protected left turns.

Some have a hand that lights up to indicate whether a person can walk or not. Some have signs that say walk or don't walk. So depending on whether our robot is walking or driving, it may have to pay attention to different signs.

The information about walking or driving must be asserted in addition to information about the status of the light. Rules can be made to cover these conditions, but they must have more than one pattern. For example, suppose we want a rule to fire if the robot is walking and if the walk-sign says walk. A rule could be written as follows:

```
(defrule take-a-walk
    (status walking)
    (walk-sign walk)
  =>
    (printout t "Go" crlf)
)
```

The above rule has two patterns. Both patterns must be satisfied by facts in the fact-list for the rule to fire. To see how this works, enter the rule and then assert the facts `(status walking)` and `(walk-sign walk)`. When you `(run)`, the program will print out `Go` since both patterns are satisfied and the rule is fired.

You can have any number of patterns or actions in a rule. The important point to realize is that the rule is placed on the agenda only if *all* the patterns are satisfied by facts. This type of restriction is called a *logical AND conditional element* in reference to the AND relation of Boolean logic. An AND relation is said to be true only if all its conditions are true.

Because the patterns are of the logical AND type, the rule will not fire if only one of the patterns is satisfied. All facts must be present before the left-hand side of a rule is satisfied and the rule is placed on the agenda.

3.3 A Question Of Strategy

The word *strategy* was originally a military term for the planning and operations of warfare. In expert systems, one use of the term strategy is in conflict resolution of activations. Now you might say, "Well, I'll just design my expert system so that only one rule can possibly be activated at one time. Then there is no need for conflict resolution." The good news is that if you succeed, conflict resolution is indeed unnecessary. The bad news is that this success proves that your application can be well represented by a sequential program. So you should have coded it in Ada, C, or Pascal in the first place and not bothered writing it as an expert system.

Jess offers two different modes of conflict resolution: *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. It's difficult to say that one is clearly better than another without considering the specific application. Even then, it may be difficult to judge which is "best." The command `set-strategy` lets you specify the conflict resolution strategy Jess uses.

3.4 Deffacts

As you work with Jess, you may become tired of typing in the same assertions from the top-level. If you are going to use the same assertions every time a program is run, you can first load assertions from a disk using a batch file. An alternative way to enter facts is by using the define facts keyword, `deffacts`. For example,

```
Jess> (clear)
TRUE
Jess> (deffacts walk
    "some facts about walking"

    (status walking) ; fact to be asserted
    (walk-sign walk) ; fact to be asserted
)
TRUE
Jess> (reset)        ; causes facts from defacts to be
                     ; asserted
TRUE
Jess> (facts)
f-0   (MAIN::initial-fact)
f-1   (MAIN::status walking)
f-2   (MAIN::walk-sign walk)
For a total of 3 facts.
Jess>
```

The required name of this `deffacts` statement, `walk`, follows the `deffacts` keyword. Following the name is an optional comment in double quotes. Like the optional comment of a rule, the `deffacts` comment will be retained with the `deffacts` after it's been loaded by Jess. After the name or comment are the facts that will be asserted in the fact-list. The facts in a `deffacts` statement are asserted using the Jess `reset` command.

The `(initial-fact)` is put in automatically by a `reset`. The fact-identifier of the initial-fact is always `f-0`. Even without any `deffacts` statements, a `reset` always will assert an `(initial-fact)`. The utility of `(initial-fact)` lies in starting the execution of a program. A Jess program will not start running unless there are rules whose left-hand sides are satisfied by facts. Rather than having to type in some fact to start things off, the `reset` command asserts it for you as well as asserting the facts in `deffacts` statements.

The `reset` has a further advantage compared to a `clear` command in that `reset` doesn't get rid of all the rules. The `reset` leaves your rules intact. Like `clear`, it removes all activated rules from the agenda and also removes all old facts from the fact-list. Giving a `reset` command is a recommended way to start off program execution, especially if the program has been run before and the fact-list is cluttered with old facts.

In summary, the `reset` does three things for facts.

1. It removes existing facts from the fact-list, which may remove activated rules from the agenda.

2. It asserts `(initial-fact)`.
3. It asserts facts from existing `deffacts` statements.

3.5 Debugging

A useful debugging command is `run` which takes an optional argument of the number of rule firings. For example, a `(run 21)` command would tell Jess to run the program and then stop after 21 rule firings. A `(run 1)` command allows you to step through a program firing one rule at a time.

Chapter 4

Variables

The type of rules that you've seen so far illustrates simple matching of patterns to facts. In this chapter, you'll learn very powerful ways to match and manipulate facts.

4.1 Let's Get Variable

Just as with other programming languages, Jess has *variables* to store values. Unlike a fact, which is *static* or unchanging, the contents of a variable are *dynamic* as the values assigned to it change. In contrast, once a fact is asserted, it's fields can only be modified by retracting and asserting a new fact with the changed fields.

The name of a variable, or *variable identifier*, is always written by a question mark followed by a symbol that is the name of the variable. The general format is

```
?<variable-name>
```

Before a variable can be used, it should be assigned a value. As an example of a case where a value is *not* assigned, try to enter the following and Jess will respond with the error message shown.

```
Jess> (clear)
TRUE
Jess> (defrule test
    (initial-fact)       ; asserted by a (reset) command
  =>
    (printout t ?x crlf)
)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Jess reported an error in routine Context.getVariable
        while executing (printout t ?x crlf)
        while executing defrule MAIN::test
```

```
        while executing (run).
  Message: No such variable x.
Jess>
```

Jess gives an error message when it cannot find a value *bound* to `?x`. The term bound means the assignment of a value to a variable. Only global variables are bound in *all* rules. All other variables are only bound *within* a rule. Before and after a rule fires, nonglobal variables are not bound and so Jess will give an error message if you try to query a nonbound variable.

4.2 Be Assertive

One common use of variables is to match a value on the left-hand side and then assert this bound variable on the right-hand side. For example, enter

```
(defrule make-quack
    (duck-sound ?sound)
  =>
    (assert (sound-is ?sound))
)
```

Now assert `(duck-sound quack)`, then `run` the program. Check the facts and you'll see that the rule has produced `(sound-is quack)` because the variable `?sound` was bound to `quack`.

Of course, you also can use a variable more than once. For example, enter the following. Be sure to do a `reset` and assert `(duck-sound quack)` again.

```
(defrule make-quack
    (duck-sound ?sound)
  =>
    (assert (sound-is ?sound ?sound))
)
```

When the rule fires, it will produce `(sound-is quack quack)` since the variable `?sound` is used twice.

4.3 What The Duck Said

Variables also are used commonly in printing output, as in

```
(defrule make-quack
    (duck-sound ?sound)
  =>
    (printout t "The duck said " ?sound crlf)
)
```

Do a `reset`, enter this rule, and assert the fact and then `run` to find out what the duck said.

More than one variable may be used in a pattern, as the following example shows.

```
Jess> (clear)
TRUE
Jess> (defrule who-dun-it
    (duck-shoot ?hunter ?who)
  =>
    (printout t ?hunter " shot " ?who crlf)
)
TRUE
Jess> (reset)
TRUE
Jess> (assert (duck-shoot Brian duck))
<Fact-1>
Jess> (run)
Brian shot duck
1
Jess> (assert (duck-shoot duck Brian))
<Fact-2>
Jess> (run)
duck shot Brian
1
Jess> (assert (duck-shoot duck)) ; missing third field
<Fact-3>
Jess> (run)
0                                ; rule doesn't fire, no output
Jess>
```

Notice what a big difference the order of fields makes in determining who shot who. You can also see that the rule did *not* fire when the single-field fact `(duck)` was asserted. The rule was not activated because no field of the fact matched the second pattern constraint, `?who`.

4.4 The Happy Bachelor

Retraction is very useful in expert systems and usually done on the right-hand side rather than at the top-level. Before a fact can be retracted, it must be specified to Jess. To retract a fact from a rule, the *fact-address* first must be bound to a variable on the left-hand side.

There is a big difference between binding a variable to the contents of a fact and binding a variable to the fact-address. In the examples that you've seen such as `(duck-sound ?sound)`, a variable was bound to the value of a field. That is, `?sound` was bound to `quack`. However, if you want to remove the fact whose contents are `(duck-sound quack)`, you must first tell Jess the *address* of the fact to be retracted.

The fact-address is specified using the *left arrow*, `<-`. As an example of fact retraction from a rule,

```
Jess> (clear)
TRUE
Jess> (assert (bachelor Dopey))
<Fact-0>
```

```
Jess> (facts)
f-0   (MAIN::bachelor Dopey)
For a total of 1 facts.
Jess> (defrule get-married
    ?duck <- (bachelor Dopey)
  =>
    (printout t "Dopey is now happily married " ?duck crlf)
    (retract ?duck)
)
TRUE
Jess> (run)
Dopey is now happily married <Fact-0>
1
Jess> (facts)
For a total of 0 facts.
Jess>
```

Notice that the `printout` prints the fact-index of `?duck`, `<Fact-0>`, since the left arrow bound the address of the fact to `?duck`. Also, there is no fact `(bachelor Dopey)` because it has been retracted.

Variables can be used to pick up a fact value at the same time as an address, as shown in the following example. For convenience, a `deffacts` has also been defined.

```
Jess> (clear)
TRUE
Jess> (defrule get-married
    ?duck <- (bachelor ?name)
  =>
    (printout t ?name " is now happily married" crlf)
    (retract ?duck)
)
TRUE
Jess> (deffacts good-prospects
    (bachelor Dopey)
    (bachelor Dorky)
    (bachelor Dicky)
)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Dicky is now happily married
Dorky is now happily married
Dopey is now happily married
3
Jess>
```

Notice how the rule fired on *all* facts that matched the pattern `(bachelor ?name)`.

4.5 It's not Important

Instead of binding a field value to a variable, the presence of a nonempty field can be detected alone using a *wildcard.* For example, suppose you're running a dating service for ducks, and a duckette asserts that she only dates ducks whose first name is Richard. Actually, two criteria are in this specification since there is an implication that the duck must have more than one name. So a plain `(bachelor Richard)` isn't adequate because there is only one name in the fact.

This type of situation, in which only part of the fact is specified, is very common and very important. To solve this problem, a wildcard can be used to fire the Richards.

The simplest form of wildcard is called a *single-field wildcard* and is shown by a question mark, `?`. The `?` is also called a *single-field constraint.* A single-field wildcard stands for *exactly* one field, as shown following.

```
Jess> (clear)
TRUE
Jess> (defrule dating-ducks
    (bachelor Dopey ?)
  =>
    (printout t "Date Dopey" crlf)
)
TRUE
Jess> (deffacts duck
    (bachelor Dickey)
    (bachelor Dopey)
    (bachelor Dopey Mallard)
    (bachelor Dinkey Dopey)
    (bachelor Dopey Dinkey Mallard)
)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Date Dopey
1
Jess>
```

The pattern includes a wildcard to indicate that Dopey's last name is not important. So long as the first name is Dopey, the rule will be satisfied and fire. Because the pattern has three fields of which one is a single-field wildcard, only facts of *exactly* three fields can satisfy it. In other words, only Dopeys with exactly two names can satisfy this duckette.

Suppose you want to specify Dopeys with exactly three names? All that you'd have to do is write a pattern like

```
(bachelor Dopey ? ?)
```

or, if only persons with three names whose middle name was Dopey,

```
(bachelor ? Dopey ?)
```

or, if only the last name was Dopey, as in the following:

```
(bachelor ? ? Dopey)
```

Another interesting possibility occurs if Dopey *must* be the first name, but only those Dopeys with two *or* three names are acceptable. One way of solving this problem is to write two rules. For example

```
(defrule eligible
    (bachelor Dopey ?)
  =>
    (printout t "Date Dopey" crlf)
)

(defrule eligible-three-names
    (bachelor Dopey ? ?)
  =>
    (printout t "Date Dopey" crlf)
)
```

Enter and run this and you'll see that Dopeys with both two and three names are printed. Of course, if you don't want anonymous dates, you need to bind the Dopey names with a variable and print them out.

4.6 Going Wild

Rather than writing separate rules to handle each field, it's much easier to use the *multifield wildcard*. This is a dollar sign followed by a question mark, `$?`, and represents *zero or more* fields. Notice how this contrasts with the single-field wildcard which must match exactly one field.

The two rules for dates can now be written in a single rule as follows.

```
Jess> (clear)
TRUE
Jess> (defrule dating-ducks
    (bachelor Dopey $?)
  =>
    (printout t "Date Dopey" crlf)
)
TRUE
Jess> (deffacts duck
    (bachelor Dicky)
    (bachelor Dopey)
    (bachelor Dopey Mallard)
    (bachelor Dinky Dopey)
    (bachelor Dopey Dinky Mallard)
)
TRUE
Jess> (reset)
TRUE
```

```
Jess> (run)
Date Dopey
Date Dopey
Date Dopey
3
Jess>
```

Wildcards have another important use because they can be attached to a symbolic field to create a variable such as `?name` or `$?name`. The variable can be a *single-field variable* or a *multifield variable* depending on whether a `?` or `$?` is used on the left-hand side. You can think of the `$` as a function whose argument is a single-field wildcard or a single-field variable and returns a multifield wildcard or a multifield variable, respectively.

As an example of a multifield variable, the following version of the rule also prints out the name field(s) of the matching fact because a variable is equated to the name field(s) that match:

```
Jess> (defrule dating-ducks
    (bachelor Dopey $?name)
  =>
    (printout t "Date Dopey " $?name crlf)
)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Date Dopey (Dinky Mallard)
Date Dopey (Mallard)
Date Dopey ()
3
Jess>
```

When you enter and run, you'll see the names of all eligible Dopeys. The multifield wildcard takes care of any number of fields. Also, notice that multi-field values are returned enclosed in parentheses.

Suppose you wanted a match of all ducks who had a Dopey somewhere in their name, not necessarily as their first name. The following version of the rule would match all facts with a Dopey in them and then print out the names:

```
Jess> (defrule dating-ducks
    (bachelor $?first Dopey $?last)
  =>
    (printout t "Date " $?first " Dopey " $?last crlf)
)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Date () Dopey (Dinky Mallard)
Date (Dinky) Dopey ()
Date () Dopey (Mallard)
```

```
Date () Dopey ()
4
Jess>
```

The pattern matches *any* names that have a Dopey anywhere in them.

Single- and multifield wildcards can be combined. For example, the pattern

```
(bachelor ? $? Dopey ?)
```

means that the first and last names can be anything and that the name just prior to the last must be Dopey. This pattern also requires that the matching fact will have *at least four* fields, since the `$?` matches *zero* or more fields and all the others must match *exactly* four.

4.7 Bound Variables

The first time a variable is bound it retains that value only within the rule, both on the left-hand side and also on the right-hand side, unless changed on the right-hand side. For example, in the rule below

```
(defrule bound
    (number-1 ?num)
    (number-2 ?num)
  =>
)
```

If there are some facts

```
f-0 (number-1 0)
f-1 (number-2 0)
f-2 (number-1 1)
f-3 (number-2 1)
```

then the rule can only be activated by the pair `f-0`, `f-1`, and the other pair `f-2`, `f-3`. That is, fact `f-0` cannot match with `f-3` because when `?num` is bound to `0` in the first pattern, the value of `?num` in the second pattern also must be `0`. Likewise, when `?num` is bound to `1` in the first pattern, the value of `?num` in the second pattern must be `1`. Notice that the rule will be activated twice by these four facts: one activation for the pair `f-0`, `f-1`, and the other activation for the pair `f-2`, `f-3`.

4.8 The Lucky Duck

Many situations occur in life where it's wise to do things in a systematic manner. That way, if your expectations don't work out you can try again systematically. One way of being organized is to keep a list. In our case, we'll keep a list of duck bachelors, with the most likely prospect for matrimony at the front. Once an ideal duck bachelor has been identified, we'll shoot him up to the front of the list as the lucky duck.

The following program shows how this can be done by adding a couple of rules to the `ideal-duck-bachelor` rule.

```
(defrule ideal-duck-bachelor
    (bill big ?name)
    (feet wide ?name)
  =>
    (printout t "The ideal duck is " ?name crlf)
    (assert (move-to-front ?name))
)

(defrule move-to-front
    ?move-to-front <- (move-to-front ?who)
    ?old-list <- (list $?front ?who $?rear)
  =>
    (retract ?move-to-front ?old-list)
    (assert (list ?who $?front $?rear))
    (assert (change-list yes))
)

(defrule print-list
    ?change-list <- (change-list yes)
    (list $?list)
  =>
    (retract ?change-list)
    (printout t "List is : " $?list crlf)
)

(deffacts duck-bachelor-list
    (list Dorky Dinky Dicky)
)

(deffacts duck-assets
    (bill big Dicky)
    (bill big Dorky)
    (bill little Dinky)
    (feet wide Dicky)
    (feet narrow Dorky)
    (feet narrow Dinky)
)
```

The original list is given in the duck `duck-bachelor-list` `deffacts`. When the program is run, it will provide a new list of likely candidates.

```
Jess> (reset)
TRUE
Jess> TRUE
Jess> (run)
The ideal duck is Dicky
List is : (Dicky Dorky Dinky)
3
Jess>
```

Notice the assertion `(change-list yes)` in the `move-to-front` rule. With-

out this assertion, the `print-list` rule would always fire on the original list. This assertion is an example of a *control fact* made to control the firing of another rule. Control facts are very important in controlling the activation of certain rules, and you should study this example carefully to understand why it's used. Another method of control is modules, to be discussed in Chapter 7.

The `move-to-front` rule removes the old list and asserts the new list. If the old list was not retracted, two activations would be on the agenda for the `print-list` rule but only one would fire. Only one will fire because the `print-list` rule removes the control fact required for the other activation of the same rule. You would not know in advance which one would fire, so the old list might be printed instead of the new list.

Chapter 5

Deftemplates

In this chapter, you will learn about a keyword called *deftemplate*, which stands for *define template*. This feature can aid you in writing rules whose patterns have a well-defined structure.

5.1 Mr. Wonderful

`deftemplate` is analogous to a record structure in a high-level language such as Pascal. That is, the `deftemplate` defines a group of related fields in a pattern similar to the way in which a Pascal record is a group of related data. A `deftemplate` is a list of named fields called *slots*. `deftemplate` allows access by name rather than by specifying the order of fields. `deftemplate` contributes to good style in expert systems programs and is a valuable tool of software engineering.

A *slot* is a named *single-slot* or *multislot*. A single-slot or simply slot contains exactly one field while a multislot contains zero or more fields. Any number of single or multislot slots may be used in a deftemplate. To write a slot, give the field name (attribute) followed by the field value. Note that a multislot slot with one value is strictly not the same as a single-slot slot. As an analogy, think of a cupboard (the multislot) that may contain dishes. A cupboard with one dish is not the same as a dish (single-slot.) However, the value of a single-slot slot (or variable) may match a multislot slot (or multislot variable) that has one field.

As an example of a `deftemplate` relation, see Table 5.1. A `deftemplate` may be defined for the relation `Prospect` as follows, where white space and comments are used for readability and explanation.

Attribute	Value
name	"Dopey"
assets	rich
age	99

Table 5.1: Attributes of a duck who might be considered a good matrimonial prospect.

```
(deftemplate Prospect   ; name of deftemplate relation
    "vital information" ; optional comment in quotes

    (slot name          ; name of field
        (type STRING)   ; type of field
        (default ""))   ; default value of field name
    (slot assets        ; name of field
        (type ATOM)     ; type of field
        (default rich)) ; default value of field assets
    (slot age           ; name of field
        (type INTEGER)  ; type of field
        (default 80))   ; default value of field age
)
```

In this example, the components of deftemplate are structured as:

- A `deftemplate` relation name
- Attributes called *fields*
- The field type, which can be any one of the allowed types: `ANY`, `INTEGER`, `FLOAT`, `NUMBER`, `ATOM`, `STRING`, `LEXEME`, and `OBJECT`.
- The default for the field value

This particular deftemplate has three single-slot slots called `name`, `assets`, and `age`.

The `deftemplate` *default values* are inserted by Jess when a `reset` is done if no explicit values are defined. For example, enter the `deftemplate` for `Prospect`, and assert it as shown.

```
Jess> (assert (Prospect))
<Fact-0>
Jess> (facts)
f-0   (MAIN::Prospect (name "") (assets rich) (age 80))
For a total of 1 facts.
Jess>
```

You can explicitly set the field values, as the following example shows.

```
Jess> (assert (Prospect (age 99) (name "Dopey")))
<Fact-1>
Jess> (facts)
f-0   (MAIN::Prospect (name "") (assets rich) (age 80))
f-1   (MAIN::Prospect (name "Dopey") (assets rich) (age 99))
For a total of 2 facts.
Jess>
```

Note that the order that the fields are typed in does not matter since these are named fields.

5.2 Bye-Bye

In general, a `deftemplate` with n slots has the following general structure:

```
(deftemplate <name>
    (slot-1)
    (slot-2)

    (slot-n)
)
```

A rule which uses the `deftemplate` follows.

```
Jess> (reset)
TRUE
Jess> (defrule matrimonial-candidate
    (Prospect (name ?name) (assets ?net-worth) (age ?months))
  =>
    (printout t "Prospect: " ?name crlf
                             ?net-worth crlf
                             ?months " months old" crlf)
)
TRUE
Jess> (assert (Prospect (age 99) (name "Dopey")))
<Fact-1>
Jess> (run)
Prospect: Dopey
rich
99 months old
Jess>
```

Notice that the default value of `rich` was used for Dopey since the `assets` field was not specified in the assert command.

If the `assets` field is given a specific value such as `poor`, the specified value for `assets` of `poor` overrides the default value of `rich` as shown in the following example about Dopey's penurious nephew.

```
Jess> (reset)
TRUE
Jess> (assert (Prospect (name "Dopey Notwonderful")
                        (assets poor)
                        (age 95)))
<Fact-1>
Jess> (run)
Prospect: Dopey Notwonderful
poor
95 months old
1
Jess>
```

A `deftemplate` pattern may be used just like any ordinary pattern. For example, the following rule will eliminate undesirable prospects.

```
Jess> (defrule bye-bye
    ?bad-prospect <- (Prospect (assets poor) (name ?name))
  =>
    (retract ?bad-prospect)
    (printout t "Bye-bye " ?name crlf))
TRUE
Jess> (run)
Bye-bye Dopey Notwonderful
1
Jess>
```

5.3 Ain't No Strings On Me

Notice that only single fields were used for the patterns in the examples so far. That is, the field values for `name`, `assets`, and `age`, were all single values. In some types of rules, you may want multiple fields. `deftemplate` allows the use of multiple values in a *multislot*.

As an example of multislot, suppose that you wanted to treat the name of the relation `Prospect` as multiple fields. This would provide more flexibility in processing prospects since any part of the name could be pattern matched. Shown following is the `deftemplate` definition using multislot and the revised rule to pattern match on multiple fields. Notice that a multislot pattern, `$?name`, is now used to match all the fields that make up the name. For convenience, a `deffacts` is also given.

```
Jess> (clear)
TRUE
Jess> (deftemplate Prospect
    (multislot name)
    (slot assets
        (type ATOM)
        (default rich))
    (slot age
        (type INTEGER)
        (default 80))
)
TRUE
Jess> (defrule happy-relationship
    (Prospect (name $?name) (assets ?net-worth) (age ?months))
  =>
    (printout t "Prospect: " $?name crlf
                             ?net-worth crlf
                             ?months " months old" crlf)
)
TRUE
Jess> (deffacts duck-bachelor
    (Prospect (name Dopey Wonderful) (assets rich) (age 99))
)
TRUE
```

```
Jess> (reset)
TRUE
Jess> (run)
Prospect: (Dopey Wonderful)
rich
99 months old
1
Jess>
```

In the output, the parentheses around Dopey's name are put in by Jess to indicate that this is a multislot value. If you compare the output from this multislot version to the single-slot version, you'll see that the double quotes around —Dopey Wonderful— are gone.

Type specification is not allowed for multislots. To specify a default value for a multislot you have to use the `create$` function to create a multifield.

```
(deftemplate Prospect
    (multislot name
        (default (create$ anonymous)))
    ...
```

5.4 What's In A Name

`deftemplate` greatly simplifies accessing a specific field in a pattern because the desired field can be identified by its slot name. The `modify` action can be used to retract and assert a new fact in one action by specifying one or more template slots to be modified.

As an example, consider the following rules which show what happens when duck-bachelor Dopey Wonderful loses all his fish buying Donald Duck posters and banana fishsplits for his new duckette, Dixie.

```
Jess> (defrule make-bad-buys
    ?prospect <- (Prospect (name $?name)
                           (assets rich)
                           (age ?months))
  =>
    (printout t "Prospect: " $?name crlf
                "rich" crlf
                ?months " months old" crlf
                crlf)
    (modify ?prospect (assets poor))
)
TRUE
Jess> (defrule poor-prospect
    ?prospect <- (Prospect (name $?name)
                           (assets poor)
                           (age ?months))
  =>
    (printout t "Ex-prospect: " $?name crlf
                                poor crlf
```

```
                                ?months " months old" crlf
                                crlf)
)
TRUE
Jess> (deffacts duck-bachelor
    (Prospect (name Dopey Wonderful) (assets rich) (age 99))
)
TRUE
Jess> (reset)
TRUE
Jess> (run)
Prospect: (Dopey Wonderful)
rich
99 months old

Ex-prospect: (Dopey Wonderful)
poor
99 months old

2
Jess>
```

The `make-bad-buys` rule is activated by a rich prospect as specified by the `assets` slot. This rule changes the assets to poor using the `modify` action. Notice that the slot `assets` can be accessed by name. Without a `deftemplate`, it would be necessary to enumerate all the fields by single variables or by using a wildcard, which is less efficient. The purpose of the `poor-prospect` rule is simply to print out the poor prospects, thus demonstrating that the `make-bad-buys` rule did indeed modify the assets.

Chapter 6

Functions

In this chapter, you will learn how to define your own functions with *deffunction*.

6.1 Doing Your Own Thing

Just like other languages, Jess allows you to define your own functions with `deffunction`. The `deffunction` is known globally, which saves you the effort of entering the same actions over and over again.

Deffunctions also help in readability. You can call a `deffunction` just like any other function. A `deffunction` may also be used as the argument of another function. A `printout` can be used anywhere in a `deffunction` even if it's not the last action because printing is a side-effect of calling the `printout` function.

The general syntax of a `deffunction` is shown following.

```
(deffunction <function-name>
    (?arg1 ?arg2 ...?argM [$?argN]) ; argument list. Last one
                                    ; may be optional multifield
                                    ; argument.
    [optional comment]

    <action-1>                      ; action-1 to action-n-1 do
    <action-2>                      ; not return a value

    <action-n-1>
    <action-n>                      ; only last action returns a
                                    ; value
)
```

The `?arg` are *dummy arguments*, which means that the names of the arguments will not conflict with variable names in a rule if they are the same. The term dummy argument is sometimes called a *parameter* in other books.

Although each action may have returned values from function calls within the action, these are blocked by the `deffunction` from being returned to the user. The `deffunction` will only return the value of the *last* action, `<action-n>`. This action may be a function, a variable, or a constant.

The following is an example of how a `deffunction` is defined to calculate the hypotenuse, and then used in a rule. Even if the variable names in the rule are the same as the dummy arguments, there's no conflict. That's why they're *dummy*, because they don't mean anything.

```
Jess> (deffunction hypotenuse     ; name
    (?a ?b)                       ; dummy arguments

    (sqrt(+ (* ?a ?a) (* ?b ?b))) ; action
)
TRUE
Jess> (defrule calculate-hypotenuse
    (dimensions ?base ?height)
  =>
    (printout t "Hypotenuse=" (hypotenuse ?base ?height) crlf)
)
TRUE
Jess> (assert (dimensions 3 4))
<Fact-0>
Jess> (run)
Hypotenuse=5.0
1
Jess>
```

Deffunctions may be used with multifield values, as the following example shows.

```
Jess> (deffunction count ($?arg)
    (length$ $?arg)
)
TRUE
Jess> (count 1 2 3 a duck "quacks")
6
Jess>
```

Chapter 7

Modules

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

7.1 Introduction

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.

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

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.

7.2 Defining Constructs In Modules

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

```
Jess> (defmodule WORK)
TRUE
Jess>
```

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.
Jess>
```

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

```
Jess> (clear)
TRUE
Jess> (get-current-module)
MAIN
Jess> (defmodule COMMUTE)
TRUE
Jess> (get-current-module)
COMMUTE
Jess>
```

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 <- (COMMUTE::Bus (route-number 76))
   (COMMUTE::have-correct-change)
   =>
   (get-on ?bus))"
Jess>
```

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

7.3 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
   (WORK::Job (salary ?s))
   (HOME::Hobby (income ?i&:(> ?i (/ ?s 2))))
   (MAIN::mortgage-payment ?m&:(< ?m ?i))
   =>
   (call-boss)
   (quit-job))"
Jess>
```

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.

7.4 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
Jess>
```

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
Jess>
```

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`.

7.4.1 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
Jess>
```

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."

7.4.2 Returning From A Rule Right-Hand-Side

If the function `return` is called from a rule's right-hand-side, it immediately terminates the execution of that rule's right-hand-side. 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 right-hand-side using `focus`, and you return from the call using `return`.

Chapter 8

Fuzzy Logic

Jess can be used to create programs that encode fuzzy operations and fuzzy reasoning. Fuzzy logic programs fit nicely into the rule based paradigm.

8.1 Introduction

In the real world there exists much *fuzzy knowledge*, i.e., knowledge that is vague, imprecise, uncertain, ambiguous, inexact, or probabilistic in nature. Human thinking and reasoning frequently involve fuzzy information, possibly originating from inherently inexact human concepts and matching of similar rather than identical experiences. In systems based upon classical set theory and two-valued logic, it is very difficult to answer some questions because they do not have completely true answers. Humans, however, can give satisfactory answers, which are probably true. Expert systems should not only give such answers but also describe their reality level. This level should be calculated using imprecision and the uncertainty of facts and rules that were applied. Expert systems should also be able to cope with unreliable and incomplete information and with different expert opinions.

8.2 Fuzziness

Fuzziness occurs when the boundary of a piece of information is not clear-cut. For example, words such as *young*, *tall*, *good*, or *high* are fuzzy. There is no single quantitative value which defines the term young when describing a fuzzy concept (or *fuzzy variable*) such as age. For some people, age 25 is young, and for others, age 35 is young. The concept young has no clean boundary. Age 1 is definitely young and age 100 is definitely not young; however, age 35 has some possibility of being young and usually depends on the context in which it is being considered. In fact an age can have some possibility of being young and also some possibility of being old at the same time (note that these are NOT probabilities and the sum of all the possibilities does not need to sum to 1.0). The representation of this kind of information is based on the concept of *fuzzy set theory*. Unlike classical set theory where one deals with objects whose membership to a set can be clearly described, in fuzzy set theory, membership of an element in a set can be partial, i.e., an element belongs to a set with a

Age	Grade Of Membership
25	1.0
30	0.8
35	0.6
40	0.4
45	0.2
50	0.0

Table 8.1: Fuzzy term *young*.

certain grade (possibility) of membership. More formally a fuzzy set F in a universe of discourse U is characterized by a membership function

$$ {}_F : U \rightarrow [0,1] $$

which associates a number ${}_F(x)$ in the interval $[0,1]$ with each element x of U. This number represents the grade of membership of x in the fuzzy set F (with 0 meaning that x is definitely not a member of the set and 1 meaning that it definitely is a member of the set). For example, the fuzzy term *young* might be defined by the fuzzy set in Table 8.1.

One might also write

$$ {}_{young}(25) = 1,\ {}_{young}(30) = 0.8,\ \ldots,\ {}_{young}(50) = 0 $$

Grade of membership values constitute a *possibility distribution* of the term *young* as applied to the fuzzy variable *age*. The Table can also be shown graphically (see Figure 8.1).

The possibility distribution of a fuzzy concept like *somewhat young* or *very young* can be obtained by applying arithmetic operations to the fuzzy set of the basic fuzzy term *young*, where the modifiers *somewhat* and *very* are associated with specific mathematical functions. For instance, the possibility values of each age in the fuzzy set representing the fuzzy concept *somewhat young* might be calculated by taking the square root of the corresponding possibility values in the fuzzy set of young (see Figure 8.2). These modifiers are often referred to as hedges. The available hedges include among others: `not`, `more_or_less`, `somewhat`, `very`, `extremely` and `slightly`.

8.3 Example

A small example will serve to introduce the basic concepts for creating fuzzy rules in Jess. In this example everything is done using only Jess code. There is quite a bit of opportunity to create code that is a mix of Jess and Java, but Jess can easily reference Java classes and this allows one to work entirely in Jess when appropriate.

The pseudocode for a rule might be

```
IF   temperature is hot
THEN pressure is low or medium
```

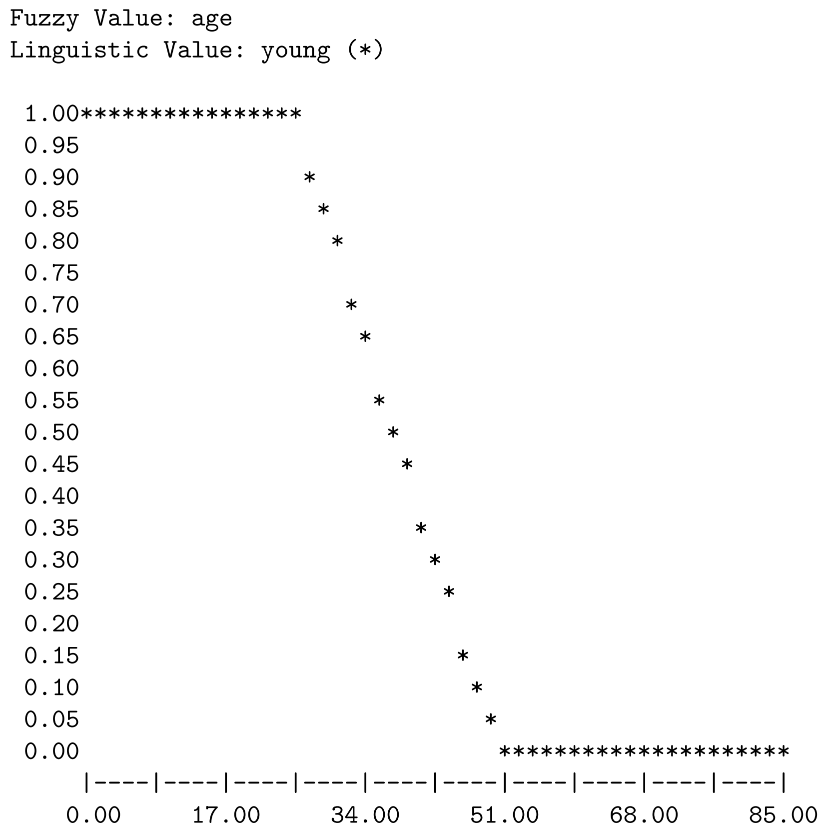

Figure 8.1: Possibility distribution of *young*.

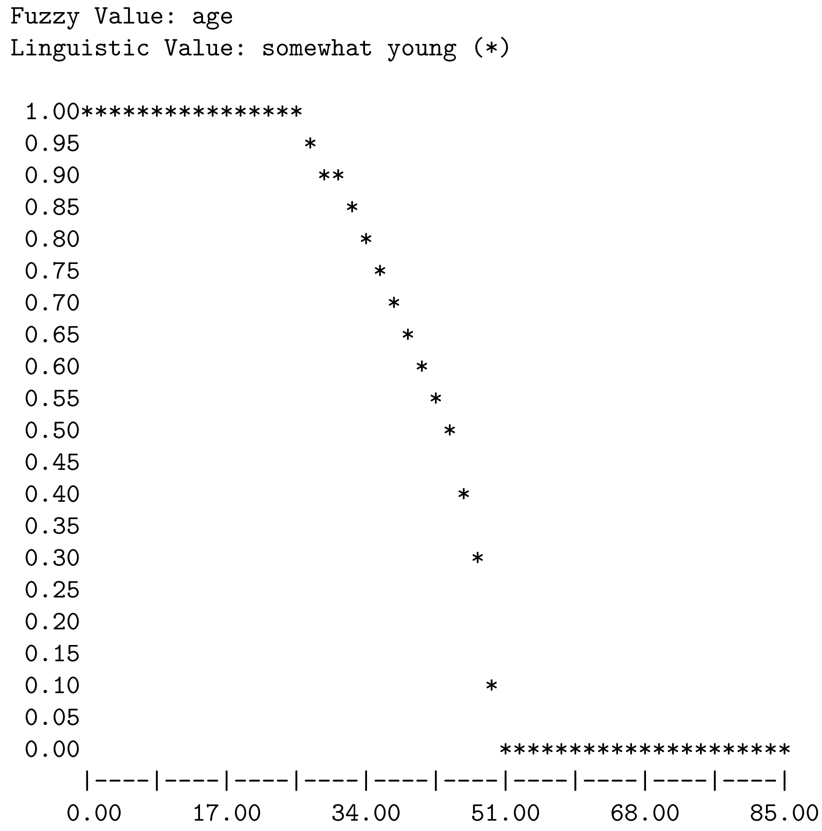

Figure 8.2: Possibility distribution of *somewhat young*.

Then by providing an input value for the temperature, in this case *temperature is very medium*, the rule can be fired and an actual conclusion is provided as the output of the rule firing.

```
; two globals to hold our fuzzy variables for temperature and
; pressure
(defglobal ?*fuzzy-temperature* =
  (new nrc.fuzzy.FuzzyVariable "temperature" 0.0 100.0 "C"))
(defglobal ?*fuzzy-pressure* =
  (new nrc.fuzzy.FuzzyVariable "pressure" 0.0 10.0
  "kilopascals"))

(defrule init
    "initialization rule that adds the terms to the fuzzy
      variables and asserts the input fuzzy value ``temperature
      is very medium''"

    ?fact <- (initial-fact)
  =>
    (retract ?fact)

    (load-package nrc.fuzzy.jess.FuzzyFunctions)

    (bind ?x-hot  (create$ 25.0 35.0))
    (bind ?y-hot  (create$ 0.0 1.0))
    (bind ?x-cold (create$ 5.0 15.0))
    (bind ?y-cold (create$ 1.0 0.0))

    ; terms for the temperature fuzzy variable
    (?*fuzzy-temperature* addTerm "hot" ?x-hot ?y-hot 2)
    (?*fuzzy-temperature* addTerm "cold" ?x-cold ?y-cold 2)
    (?*fuzzy-temperature* addTerm "veryHot" "very hot")
    (?*fuzzy-temperature* addTerm "medium"
      "not hot and (not cold)")

    ; terms for the pressure fuzzy variable
    (?*fuzzy-pressure* addTerm "low"
      (new nrc.fuzzy.ZFuzzySet 2.0 5.0))
    (?*fuzzy-pressure* addTerm "medium"
      (new nrc.fuzzy.PIFuzzySet 5.0 2.5))
    (?*fuzzy-pressure* addTerm "high"
      (new nrc.fuzzy.SFuzzySet 2.0 5.0))

    ; add the fuzzy input ``temperature is very medium''
    (assert (temperature
      (new nrc.fuzzy.FuzzyValue ?*fuzzy-temperature*
      "very medium")))
)

(defrule temperature-hot-pressure-low-or-medium
```

```
    "if temperature hot then pressure low or medium"

    (temperature ?temperature&:(fuzzy-match ?temperature "hot"))
  =>
    (assert (pressure
      (new nrc.fuzzy.FuzzyValue ?*fuzzy-pressure*
      "low or medium")))
)

(defrule do-the-printing
    "print some interesting things"

    (temperature ?temperature)
    (pressure ?pressure)
  =>
    (bind ?fuzzy-values (create$
      (new nrc.fuzzy.FuzzyValue ?*fuzzy-temperature* "hot")
      ?temperature))
    (printout t (call nrc.fuzzy.FuzzyValue plotFuzzyValues "*+"
      0.0 50.0 ?fuzzy-values) crlf)
    (printout t (call (new nrc.fuzzy.FuzzyValue ?*fuzzy-pressure*
      "low or medium") plotFuzzyValue "*") crlf)
    (printout t (?pressure plotFuzzyValue "*") crlf)
)
```

The first part is pretty straight forward, create the fuzzy variables we need to represent the concepts of temperature and pressure. We store these in global variables.

A *fuzzy variable* defines the language that will be used to discuss a fuzzy concept such as temperature, pressure, age, or height. The class `FuzzyVariable` is used to create instances of a fuzzy variable, providing a *name* (for example, `temperature`), the *units* of the variable if required (for example, degrees `C`), the *universe of discourse* for the variable (for example a range from 0 to 100), and a set of *primary fuzzy terms* (like `hot`, `cold` and `warm`) that will be used when describing the specific fuzzy concepts associated with the fuzzy variable. The name and units are strings that are mainly used for textual display in a program. The universe of discourse defines a set of upper and lower bounds for the values of the fuzzy sets used to describe the concepts of the fuzzy variable. All numbers involved in fuzzy reasoning have to be of type real.

We have an initialization rule that adds the appropriate terms to the fuzzy variables so we can describe our concepts. We add many more terms than we need for the example but it does show various ways to define the terms. Nothing too weird here or hard to understand if you check out the `addTerm` method for a `FuzzyVariable` in the API documentation and you know that the `load-package` function loads some necessary user functions (such as `fuzzy-match`) that support fuzzy capabilities.

The *fuzzy terms* are described using a term name such as `hot`, along with a *fuzzy set* that represents that term. The fuzzy variable terms along with a set of system supplied and user defined fuzzy modifiers, as well as the operators `and` and `or` (fuzzy set intersection and union respectively) and the left and

right parentheses provide the basis for a grammar that allows one to write *fuzzy linguistic expressions* that describe fuzzy concepts in an english-like manner. For example,

```
(very hot or warm) and slightly cold
```

consists of the terms `hot`, `warm` and `cold`, along with the fuzzy modifiers `very` and `slightly`. These expressions are used when defining fuzzy values, the specific fuzzy concepts that are appropriate for the problem at hand. `FuzzyValue` associates a fuzzy set with a fuzzy variable to describe the fuzzy concept. The fuzzy sets can be described using the linguistic expressions definable for the fuzzy variable and restricted by the universe of discourse for the variable. The fuzzy sets can also be described using an instance of a `FuzzySet` object or by sets of `x`, `y` values.

The last thing done in the `init` rule is to assert a temperature fact that has a fuzzy value in it:

```
(temperature (new nrc.fuzzy.FuzzyValue ?*fuzzy-temperature*
  "very medium"))
```

We used an ordered fact in this case but slots in unordered facts can also hold fuzzy values.

The next rule is the heart of the program. Notice that it is quite compact and not too difficult to read the intent. So, what is happening here? When a temperature fact is asserted that has a fuzzy value in it, the `fuzzy-match` will compare the fuzzy value in the fact to the fuzzy value `hot`. In this case the `fuzzy-match` function will succeed since there is overlap between the two fuzzy values. Internally Jess will remember the pair of fuzzy values that matched for use when the rule fires and tries to assert other facts with fuzzy values. Using all of the remembered antecendent/input pairs that matched on the LHS of the rule, a fuzzy rule is constructed (if it has not already been constructed by another assert on the RHS of the rule). The fuzzy values identified in the fact being asserted are added as outputs for the fuzzy rule and it is fired, producing the actual fuzzy values that will be placed in the fuzzy value positions of the asserted fact. In our case, the antecedent and the input fuzzy values are *temperature hot and temperature very medium.* They overlapped and were remembered and used to construct the fuzzy rule. The output fuzzy value from the assert function, *pressure low or medium*, is added to the rule and it is fired. The output fuzzy value is then placed into the actual fact that is asserted. The actual results are shown below. This might sound complicated but the Jess user does not need to be aware of all of these details. The user simply needs to know that it is necessary to use the `fuzzy-match` function on the LHS of a rule to match fuzzy values and that assertions done on the RHS of a rule that have fuzzy values in them will automatically be adjusted to account for fuzzy matching on the LHS of the rule.

Of course it is possible to create rules with many fuzzy value patterns on the LHS and many fuzzy values asserted in fact on the RHS. These can be mixed with non-fuzzy facts and various tests.

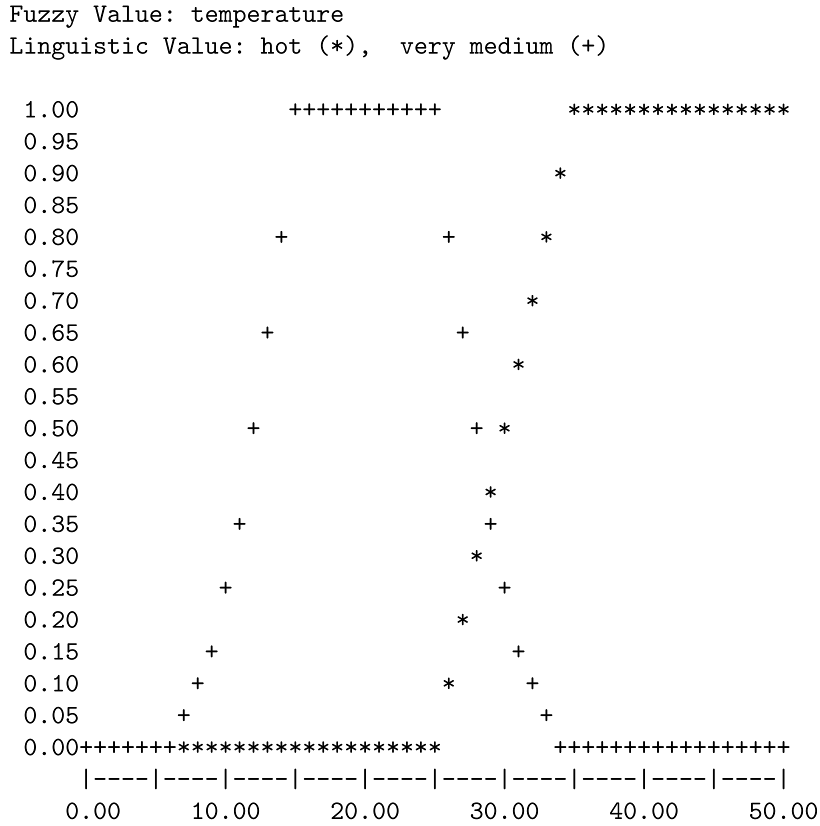

Figure 8.3: The antecedent (hot) and the input (very medium) fuzzy values.

8.4 Example Results

The Jess rules in the example would produce the Figures 8.3–5 as output. Note that when displaying these text plots it is necessary to use a mono-spaced (non-proportional) font.

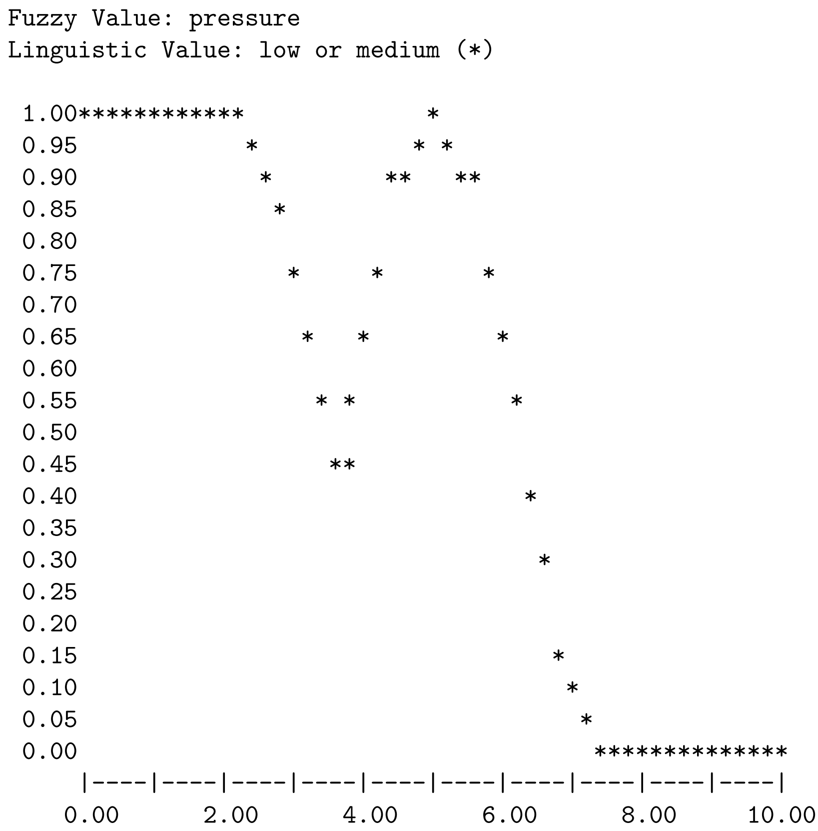

Figure 8.4: The conclusion fuzzy value.

```
Fuzzy Value: pressure
Linguistic Value: ??? (*)
```

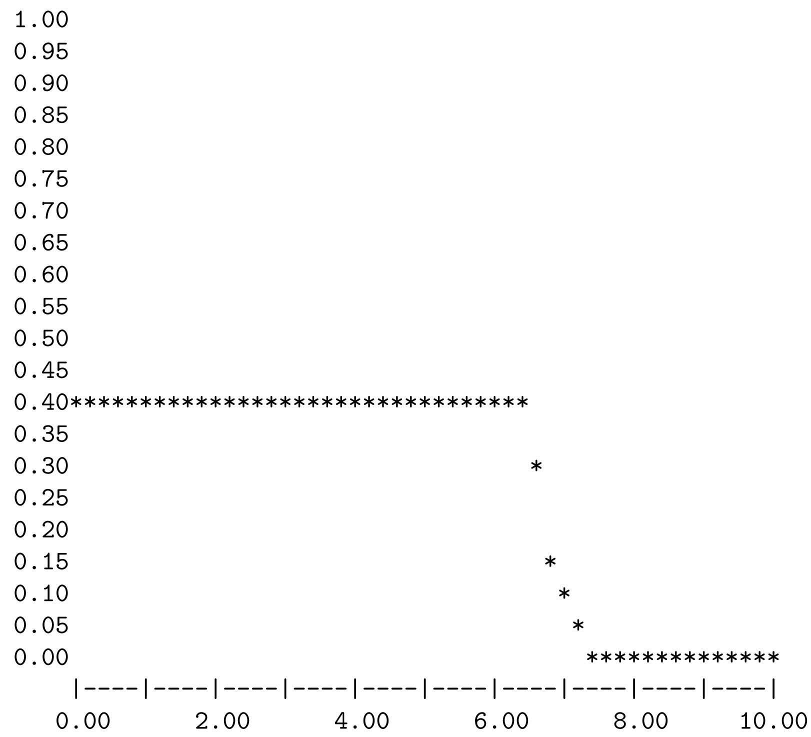

Figure 8.5: The output using MamdaniMinMaxMinRuleExecutor.

Bibliography

- Giarratano, J.C. (1998, August 5). "CLIPS User's Guide, Version 6.10".

- Friedman-Hill, E.J. (2001, December 7). "Jess, The Expert System Shell for the Java Platform, Version 6.0". Retrieved March 14, 2002 from the World Wide Web: http://herzberg.ca.sandia.gov/jess/docs/60/.

- Orchard, R.A. (2001, June). "NRC FuzzyJ Toolkit for the Java™ Platform, User's Guide, Version 1.2". Retrieved April 11, 2002 from the World Wide Web: http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyJDocs/index.html.

14. Can Jess do backwards chaining? 15. How do I loop over all the facts my rule LHS matches? 16. When I try one of the code examples in the Jess manual, I get an exception. 17. I wrote a rule with a function call on the LHS, and it never fires. Why? Basic JESS Tutorial. Pete Barnum . Intro to Jess: JESS, which stands for Java Expert System Shell, is a powerful system that allows the solution of rule-based problems. Jess is written entirely in Java and as a result, it is possible to run any Java code inside of it, with the exception of defining new classes. Jess uses a Common LISP (CLISP) type syntax and familiarity with CLISP is recommended. This tutorial will cover basic commands and usage.