Jess

The Java Expert System Shell

April 26, 2007

Martin Strauss
martin@ockle.org
Universität des Saarlandes
Matrikelnummer: 2512940

Seminar "AI Tools"
Wintersemester 2006/2007
Betreuer: Michael Kipp

Abstract

This paper introduces Jess, the Java Expert System Shell. Jess is an interpreter for the Jess Language, a rule-based language for specifying expert systems.

This paper first introduces the concept of expert systems and production systems, as well as the typical architecture of such a system. Then it presents a thorough outline of the Jess Language: the syntax of facts, rules and functions, allowed constructs, built-in functions and the module structure. It also presents the execution cycle of the Jess engine, as well as a number of methods to influence the default progress of this cycle. Due to time and space constraints, Jess's interaction with Java will not be covered; a second paper on Jess by Jens Haupert will contain this information (among others).

Finally, this paper introduces the example of a tax file advisor software (for example in a kiosk setting) to illustrate the capabilities of Jess and the concepts presented.

Contents

Chapter 1

Introduction

Jess, the Java Expert System Shell, is an interpreter for the Jess Language, a rule-based language for specifying expert systems. Architecturally, Jess is a production system executing a rule-based program; thus, the Jess language is a declarative (rather than imperative) language.

The Jess engine can be invoked as an interactive interpreter, where Jess language strings can be typed into a shell and invoked in real-time, or in batch mode, where one or multiple files of Jess code can be executed at once. The Jess engine is implemented in Java, and as well as the shell or interpreter mode, it can also be invoked from Java code at runtime. Jess code is able to call other Java code, or be executed in a Java object.

This paper will first introduce the concept of an expert system, using a simple logic puzzle as an example; and then it will introduce production systems and how they can be used to solve problems such as the simple example. After this motivation, chapter 3 will outline the Jess Language: its syntax, symbols, values, variables, lists, functions, conditional elements, facts, rules and constructs, as well as introducing a number of useful built-in functions and constructs.

Chapter 4 will detail the execution cycle of the interpreter. We will look at the steps involve in executing a rule-based program, and how Jess implements these steps. We will also look at mechanisms Jess provides to affect the default implementation of these steps in Jess; and we will also introduce the concept of modules, both as a way to influence control flow in a Jess program and as a system of namespaces and modularisation of code.

Finally chapter 5 will introduce a simple expert system: a tax form advisor application intended to run on a kiosk and advise visitors about which tax forms they need to fill in, based on the answers visitors give to a number of questions asked by the system prior to its recommendation. This example will serve as an example Jess program, and illustrate the capabilities of Jess described in the rest of the paper.

This paper will not cover the interaction of Jess with Java; a second paper on Jess by Jens Haupert will cover this topic (among others). For more information on Jess the best resource is the online manual, [Hill(2006)].

Chapter 2

Expert systems

Jess stands for the "Java Expert System Shell". First of all, what is an "expert system"? And what is a production system?

2.1 An example problem

Consider the problem in figure 2.1. How do we solve it? You can buy books full of this sort of problem; they are usually called "logic puzzles" or something similar. They come accompanied with a n-dimensional table of combinations of variables (the colour of Bob's pants, for instance; or the name of the golfer who is second in line). To solve the puzzle, the reader enters the facts from the clues into this table (using ticks and crosses, for example), and then by elimination the table can be filled, producing the answer. So we have a specific, repeatable algorithm for solving such a problem. But what if we only had to specify the clues from the problem in some standard way, and could let a computer solve it for us?

Specifically, we would like to transform the clues in figure 2.1 into the facts in figure 2.2 (for simplicity, these facts are in first-order logic: we will present the JESS syntax in chapter 3); and then ask a program to tell us for each value of $g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\}$, for which values of $p \in \text{pants}, c \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\}$ $\text{position}(g, p)$ and $\text{pants}(g, c)$ are true.

We call such a system an "expert system": that is, an expert system is a system that can reason about the world, and take appropriate actions based on some kind of knowledge about the world. Since the program is specified using facts and rules, and letting a reasoning engine operate on them, an expert system is also called a "rule-based system".

Since expert systems are programmed by specifying facts about the world rather than steps in an algorithm, they are a kind of declarative programming. Declarative programming is much more intuitive than imperative programming in many situations, in particular situations requiring this sort of reasoning; here, we no longer need to specify an algorithm for solving logic puzzles, instead we

1. A foursome of golfers is standing at a tee, in a line from left to right. Each golfer wears different colored pants; one is wearing red pants. The golfer to Fred's immediate right is wearing blue pants.
2. Joe is second in line.
3. Bob is wearing plaid pants.
4. Tom isn't in position one or four, and he isn't wearing the hideous orange pants.

In what order will the four golfers tee off, and what color are each golfer's pants?

Figure 2.1: A logic puzzle, taken from [Hill(2003)]

1.
   - $\forall g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{golfer}(g)$
   - $\forall g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \exists p \in \{1, 2, 3, 4\} : \text{position}(g, p)$
   - $\forall p \in \{1, 2, 3, 4\} \exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{position}(g, p)$
   - $\forall g, h \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \forall p, q \in \{1, 2, 3, 4\} : \text{position}(g, p) \wedge \text{position}(h, q) \rightarrow (g = h) \vee (p \neq q)$
   - $\forall g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \exists c \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\} : \text{pants}(g, c)$
   - $\forall c \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\} \exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{pants}(g, c)$
   - $\forall g, h \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \forall c, d \in \{\text{Plaid}, \text{Red}, \text{Blue}, \text{Orange}\} : \text{pants}(g, c) \wedge \text{pants}(h, d) \rightarrow (g = h) \vee (c \neq d)$
   - $\exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} : \text{pants}(g, \text{Red})$
   - $\exists g \in \{\text{Joe}, \text{Bob}, \text{Tom}, \text{Fred}\} \exists p, q \in \{1, 2, 3, 4\} : \text{position}(\text{Fred}, p) \wedge \text{position}(g, q) \wedge q = p + 1 \wedge \text{pants}(g, \text{Blue})$
2. $\text{position}(\text{Joe}, 2)$
3. $\text{pants}(\text{Bob}, \text{Plaid})$
4. $\text{position}(\text{Tom}, p) \wedge p \neq 1 \wedge p \neq 4$
5. $\text{pants}(\text{Tom}, t) \wedge t \neq \text{Orange}$

Figure 2.2: The facts from the logic puzzle in figure 2.1 as first order logic facts

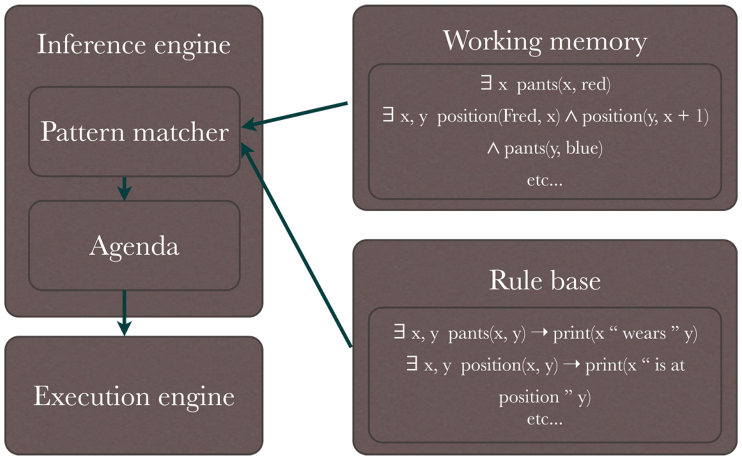
Inference engine
Pattern matcher
Agenda
Execution engine
Working memory
∃ x pants(x, red)
∃ x, y position(Fred, x) ∧ position(y, x + 1)
∧ pants(y, blue)
etc...
Rule base
∃ x, y pants(x, y) → print(x " wears " y)
∃ x, y position(x, y) → print(x " is at position " y)
etc...

Figure 2.3: The architecture of a typical rule based system

only need to formally state the puzzle, allowing us to focus on the problem itself. Admittedly, such a logic puzzle can be solved using a pretty simple algorithm; but if you've ever found yourself buried in complex if/then/else constructs covering lots of complicated and varying cases, you can appreciate how much easier it would be to just specify a list of rules and let the system do all the hard work for you. Of course there are other advantages to this style of coding as well: for a problem that consists mainly of rules and facts (such as this logic puzzle), the format of rule-based code is much more closely related to the format of the problem, and is therefore much easier to read and also easier to maintain or change if the problem changes. Also, separating the declaration of the problem from the algorithm for solving it allows the code to be much more easily understood by non-programmers: thus this style of programming is ideal for expert systems produced by experts in a field (which is not necessarily software engineering).

2.2 Typical architecture

A rule based system generally follows the architecture in figure 2.3; this is the architecture of Jess. The working memory stores facts about the world; it is also sometimes called the "fact base".

The rule base stores the system's rules. The rules and the initial facts (together with some functions) form a Jess "program". The inference engine is the "black box" that performs reasoning over the facts and rules; the left hand side of figure 2.3 is called the "rule engine". Inside the rule engine, the pattern matcher selects rules that are applicable given the facts in the fact base, and activates these rules - that is, places them on the agenda. Then the execution engine fires the rules on the agenda in a particular order. The agenda can also be called the "conflict set", since all the rules on the agenda are applicable now, and are therefore in conflict; then the algorithm by which the execution engine decides on an order in which to fire the rules is called “conflict resolution”.

If we have a standard way of specifying rules and facts for the rule base and the working memory then the rule engine and the execution engine together form a standalone system. Jess is such a system. In fact, Jess is both a rule engine and a language; the Jess engine is implemented in Java, and the Jess language is Lisp-like in appearance (in fact Jess started life as a re-implementation in Java of another rule-based system called CLIPS, so the syntax of Jess is reminiscent of CLIPS and may seem familiar if you've had experience with CLIPS).

The golfers puzzle in figure 2.1 was a simplistic example of a rule-based program: all the information was already present in the facts, and the program simply had to explicitly print out what we already implicitly know. A more sophisticated system can have more complicated and powerful inference rules. As an example of such a system, let us consider a tax form advice system, that asks questions about a user's financial situation and decides which tax forms that user needs to fill out. We will use examples from this domain throughout the paper, and at the end develop this into a complete application.

Chapter 3

The Jess language

3.1 Jess at a glance

The Jess language is mostly a declarative language - that is, a Jess program is made up of a set of statements about the world, rather than a list of commands to execute. In fact, since Jess is a rule-based system, these statements about the world are either facts, or rules.

3.2 Syntax

On the surface, Jess has a very LISP-like syntax: every component (facts, rules, function calls, etc) takes the form of a list. Also, like in many languages, whitespace is ignored.

3.3 Symbols

The basic unit of Jess is the symbol. These perform a similar function to identifiers in Java - they can represent values or variables. The following characters are valid characters for a symbol:

```
a-z A-Z 0-9 $ * . = + / < > _ ? #
```

However, a symbol may not begin with a number, or with the characters `$`, `?`, `&`, and `=` : these characters have special meanings which will be detailed below. Also, symbols are case sensitive; thus `Question` is a different symbol from `question`. A number of symbols are reserved and have special meaning in the Jess language:

- `nil` is a special symbol that represents the absence of a value, rather than a value itself; it corresponds to Java's `null`.

- `TRUE` and `FALSE` are the Boolean true and false values

In addition, some other symbols are special only in a particular context; for example, `crlf` is transformed into a newline character when it is output as part of a string.

3.4 Values

Jess values can be either symbols, numbers, or strings, or lists of symbols, numbers or strings. Numbers can be either integers or doubles. Unlike Java, a string containing newlines, tab characters, etc. can be specified by simply typing those characters in the program; thus the Java escapes for these characters do not work in Jess (`\n` produces the characters \ and `n`, for example); the exception is `\"`, since the quotation mark is used to enclose a string and thus needs to be escaped within a string. Any line beginning with a semicolon is considered a comment.

3.5 Variables

Variables in Jess are denoted by strings beginning with a question mark (which is why, as we saw above, symbols cannot begin with the character `?`). Despite the fact that Jess values have types, Jess variables are untyped. Generally only alphanumeric characters, dashes and underscores are used in a variable name, although other punctuation marks are allowed.

To assign a value to a variable, Jess provides the `(bind)` function, which has the syntax `(bind <variable> <value>)`, for example:

```
(bind ?expl "Unreimbursed employee expenses")
```

A multifield is a special kind of variable that is almost identical to normal variables except in the argument list when defining a function, and on the left-hand-side of rules. In these situations, a standard variable would match only a single value, whereas a multifield allows us to specify that we are allowing one or more values. A multifield is denoted by the initial characters `$?` (instead of just `?`), for example `$?forms`.

The `(reset)` function clears all non-global variables. Global variables are persistent variables that are not cleared by `(reset)` (although they can be optionally reset to their default values). A variable is marked as global by enclosing its name in asterisks, for example `?*question*`. A global variable is defined using the `(defglobal)` function, which takes the form `(defglobal variable = default-value)`, for example:

```
(defglobal ?*x* = 3)
```

```
(deffunction ask-user (?question ?type)
  "Ask a question, and return the answer"
  (bind ?answer "")
  (while (not (is-of-type ?answer ?type)) do
         (printout t ?question " ")
         (if (eq ?type yes-no) then
           (printout t "(yes or no) "))
         (bind ?answer (read)))
  (return ?answer))
```

Figure 3.1: An example of a Jess function

3.6 Lists

A list is a ordered sequence of values. Although all snippets of Jess code look like lists (as in LISP), Jess differs from LISP in that they are not in fact interpreted as lists - by default, a "list" typed into the Jess prompt is considered a function, with the head of the list the functor and the body its arguments. Also, other data types such as rules do not obey the same syntax as lists - in fact, to define a list you need to explicitly create a list data type. Lists are defined using the `(create$)` function; usually they are then immediately bound to a variable so we can access them again later, for example:

```
(bind ?something (create$ a b c d e))
```

In addition to this function, Jess provides other list handling functions such as `(nth$)` (which returns the nth element of a list), `(first$)` (which returns the first element of a list), and `(rest$)` (which returns all of the list except the first element). Note that lists in Jess can't be nested; Jess flattens nested lists into a single list, so that the list `(a b c (d e) f)` is the same as (and is in fact represented as) the list `(a b c d e f)`.

3.7 Functions

When you write a simple list at the Jess prompt, Jess treats it as a function call: for example, `(eq 3 5)` calls the function `(eq)` which tests if its two arguments (in this case `3` and `5`) are equal. To define functions, Jess provides the function `(deffunction)`, which has the syntax `(deffunction <name> (<parameter>*) [<comment>] <expression>*)`; an example is in figure 3.1

Here we could use a multifield in the parameter list, to specify that this function accepts "one or more" parameters (rather than an exact number).

We have already seen that functions and variables are defined using function calls; we will soon see that facts and rules are also defined this way. In addition, Jess provides a number of useful built-in functions for diagnostic purposes, in particular:

- `(facts)` lists all the facts currently in working memory;
- `(rules)` lists all the rules currently in the rule base;
- `(watch)` turns on some debug output in Jess. This has the syntax `(watch <what>)`; you can watch `facts`, `rules`, `focus`, or `all`; debug output is then produced when the `watch`-ed items change. To turn watch off again, Jess provides the function `(unwatch)` with the same syntax as `watch`.

3.8 Facts

Now we have seen the lower-level foundations of the language, we should turn to the actual components of a rule-based system: facts and rules. Facts in Jess contain a "head" and optionally one or more "slots", and are either "ordered" or "unordered" facts. In an unordered fact, the slots are labelled and thus can come in any order (hence "unordered"); in an ordered fact the slots are unlabelled and get their meaning from their order (hence "ordered"). Examples of ordered and unordered facts (respectively):

```
(answer income 45000)
(answer (ident income) (text 45000))
```

Because ordered facts can quickly become difficult to keep track of, it is usual to use unordered facts for all facts with more than one slot. Obviously we need to somehow define the format our unordered facts will take; similarly to functions, Jess gives us a function `(deftemplate)` that defines an unordered facts' slots, with the syntax `(deftemplate <name> [<comment>] (slot|multislot <slotname> (<slotoptions>)*)*)`, for example:

```
(deftemplate answer
  (slot ident)
  (slot text))

(deftemplate user
  (slot income (default 0))
  (slot dependents (default 0)))
```

There are a number of different options that a slot can have; the most useful (seen here) is the `default` option, which specifies the default value a slot should take if we ever do not specify a value when asserting this fact. Alternatively, we can declare a `multislot` instead of a `slot`, which specifies that this slot will match a multifield rather than a simple variable, and can accept one or more values (like a multifield).

Facts and functions live in the working memory in Jess. Facts are added to our knowledge by the `(assert)` function, and removed from our knowledge by `(retract)`. Rules, functions and our initial facts form the Jess program.

We can clear Jess's working memory with the function `(clear)`. This resets Jess to its initial state, wiping the facts, functions and rules from working memory - that is, it deletes our program. Thus this is not something we want to do often: in fact, it is mainly used to swap out an entire program before loading a different one. Similar to this function is the `(reset)` function (which we already saw cleared all non-global variables). In fact `(reset)` resets our program to its initial state, clearing non-global variables, setting global variables to their default values, putting only our initial facts into working memory, and leaving the rules and functions intact. Thus our Jess program is ready to run from the beginning after a call to `(reset)`.

When we first start Jess, before loading any files, a (compulsory) initial call to `(reset)` sets the working memory to contain only one fact: `(initial-fact)`. We can make the initial state contain other facts as well as `(initial-fact)` by using the `(deffacts)` function. `(deffacts)` takes as an argument a number of facts, and thereafter every call to `(reset)` resets the working memory to contain exactly those facts. The syntax of `(deffacts)` is `(deffacts <name> [<comment>] <fact>*)`. An example is given in figure 3.2

3.8.1 Conditional Elements

Now we are able to specify facts, variables and values. However, in order to build up more complex logical statements we need to be able to combine these. Jess provides the following conditional elements for this purpose:

- `and`
- `or`
- `not`
- `exists`

which correspond to the Boolean operators "and", "or", "not" and the existential quantifier respectively. If the existential quantifier is absent from a logical expression, Jess considers that expression to be implicitly universally quantified; correspondingly, `(exists ?a)` is implemented internally as `(not (not ?a))`.

In addition to the Boolean operators, two more conditional elements are necessary due to the way rules are defined. Sometimes you want to call a function and evaluate its result as part of a logical expression: the conditional element `test` calls a function and evaluates to `TRUE` if the function returns anything except `FALSE`. The other conditional element is the `logical` element, which is required for particular kinds of rules and will be elaborated below.

```
(deffacts question-data
  "The questions the system can ask."
  (question (ident income) (type number)
            (text "What was your annual income?"))
  (question (ident interest) (type yes-no)
            (text "Did you earn more than $400 of
                taxable interest?"))
  (question (ident dependents) (type number)
            (text "How many dependents live with
                you?"))
  (question (ident childcare) (type yes-no)
            (text "Did you have dependent care
                expenses?"))
  (question (ident moving) (type yes-no)
            (text "Did you move for job-related
                reasons?"))
  (question (ident employee) (type yes-no)
            (text "Did you have unreimbursed employee
                expenses?"))
  (question (ident reimbursed) (type yes-no)
            (text "Did you have reimbursed employee
                expenses, too?"))
  (question (ident casualty) (type yes-no)
            (text "Did you have losses from a theft
                or an accident?"))
  (question (ident on-time) (type yes-no)
            (text "Will you be able to file on
                time?"))
  (question (ident charity) (type yes-no)
            (text "Did you give more than $500 in
                property to charity?"))
  (question (ident home-office) (type yes-no)
            (text "Did you work in a home office?")))
```

Figure 3.2: An example of the use of `deffacts` in Jess

```
(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it and
    assert the answer"
  (declare (auto-focus TRUE))
  (MAIN::question (ident ?id) (text ?text) (type
    ?type))
  (not (MAIN::answer (ident ?id)))
  ?ask <- (MAIN::ask ?id)
  =>
  (bind ?answer (ask-user ?text ?type))
  (assert (answer (ident ?id) (text ?answer)))
  (retract ?ask)
  (return))
```

Figure 3.3: An example of a Jess rule

3.9 Rules

Now we come to the core component of our rule-based system: rules. In Jess a rule takes the form of a left-hand side (consisting of facts) and a right-hand side (consisting of a sequence of function calls), separated by the characters `=>`. The Jess engine attempts to match each rule's left-hand side against facts in the working memory, and if it can successfully match the left-hand side of a rule it executes the functions in the rule's right-hand side. Rules are defined using the `(defrule)` function, which has the syntax `(defrule <name> [<comment>] <fact>* => <function>*)`; the sequence `<fact>* => <function>*` is the actual rule. An example of a rule is shown in figure 3.3.

The slots in the facts in the left hand side of the rule don't have to be specified exactly; if we specify variables instead of values here, then the rule will match any fact that has a value in that slot, and the variable will be set to the value of that slot. We can then access this value on the right hand side of the rule using that variable. If the variable already has a value, however, then using the variable is the same as using its value.

Sometimes it may be useful to be able to refer to not just slot values but the entire fact on the right hand side of a rule. Jess allows us to bind a fact to a variable using the `<-` symbol:

```
?ask <- (ask ?id)
=>
(retract ?ask)
```

Sometimes we may wish to place conditions on slot values, for example to allow a rule to match facts with slots containing a range of values rather than

```
(answer (ident income) (text ?i&:(< ?i 50000)))
(answer (ident dependents) (text ?d&:(eq ?d 0)))
=>
(assert (MAIN::ask interest))
```

Figure 3.4: Using the `&:` characters to restrict the values a slot may take

```
(logical (answer (ident childcare) (text yes)))
=>
(assert (recommendation (form 2441)
(explanation "Child care expenses")))
```

Figure 3.5: Using `logical` to assert a logical dependency

any value. Jess allows us to place such conditions using the `&:` symbol, as in figure 3.4

Now we can introduce the final conditional element mentioned (but not explained above: `logical`. This asserts a "logical dependency" between the left hand side and the right hand side of a rule; specifically, a rule marked as `logical` is not only fired when the left hand side matches facts in the fact base, but also any facts asserted by the rule's right-hand-side are retracted when the left hand side no longer matches, as in figure 3.5.

without the `logical` element, this rule would merely assert the fact `(recommendation (form 2441) (explanation "Child care expenses"))` every time the fact `(answer (ident childcare) (text yes))` appeared in the knowledge base; after adding `logical`, this rule will also retract the `recommendation` fact every time the `answer` fact disappears from the knowledge base. We say that the fact `recommendation` is "logically dependent" on the fact `answer`.

Chapter 4

Execution cycle of Jess

Now that we have seen the Jess language, we can inspect the execution cycle of the Jess engine more closely. The typical execution cycle of an expert system (and the one used by Jess) consists of a number of steps:

1. Match facts from the fact base against the left hand sides of the rules in the rule base; move matched rules onto the agenda

2. Order the rules on the agenda according to some conflict resolution strategy

3. Execute the right hand sides of ("fire") the rules on the agenda in the order decided by step 2

Every time the command `(run)` is entered at the Jess prompt, Jess performs these steps, matching each rule against each fact once, until there are no more combinations of rules and facts to attempt matching. By default, Jess uses a "first matched, first executed" policy to order the agenda; however there are a number of ways in which the programmer can influence the conflict resolution.

4.1 Salience

Every rule in Jess has a "salience" value, which specifies how urgently a rule should be fired. Rules with a higher salience value are fired earlier than rules with a lower salience value. The programmer can manually set a rule's salience value as an option to the `defrule` function. However, this is not purely declarative programming - the programmer is telling the program in what order to execute the statements! One of the advantages to declarative programming is precisely that the programmer doesn't need to make an effort to specify this sort of information; if a program requires much of this sort of firm control on order of execution, perhaps an imperative language would be better suited to the task.

4.2 Modules

Jess does, however, provide a less intrusive method to influence the order of a program's execution: it allows us to partition the rule base and fact base into named modules. Thus we can group together rules and facts that together operate on the same stage of the program's execution. Jess allows us to give "focus" to specific modules, thus temporarily enabling and disabling groups of rules and facts; but we are still letting the rule engine's own conflict resolution decide which rules from this module to fire.

By default, all Jess rules and facts live in the module `MAIN`. To define a new module, Jess provides the function `(defmodule)`, with the syntax `(defmodule <name>)`:

```
    (defmodule ask)
```

Focus movement between modules is implemented using a stack; modules are pushed onto the stack, and then popped off and processed one by one. To manipulate the focus stack, we have several functions: `(focus <modulename>*)` pushes one or more modules onto the focus stack; `(clear-focus-stack)` clears the stack; `(pop-focus)` pops the next module off the stack and gives it focus; and `(get-current-module)` gets the current module. In addition, `(defrule)` takes an option `(declare auto-focus TRUE)`, which tells Jess to always try matching that rule, and if it matches to move focus to that rule's module. The counterpart to this is the function `(return)`, which returns focus to the module from which it was 'stolen' by `auto-focus`. This allows the definition of "global" rules that can fire no matter what module currently has focus; it also allows the focus of the program to "sidestep" temporarily out of one module for the purpose of firing such rules, without changing focus fully. Alternatively, we can think of this as similar to a subroutine call: a rule can be "called" (when its facts match), and then focus returns to where we were before firing that rule. An example of the usage of `auto-focus` and `(return)` is in figure 4.1.

For example, in the tax advisor example mentioned earlier, we would separate the rules into "ask questions" rules and "make recommendations" modules, so we can have the system ask all its questions at once first, and then make all its recommendations together at the end.

4.3 Namespaces

Once we have organised rules and facts into modules, we have implicitly defined some namespaces for our rules. A name qualified with a namespace has the syntax `<namespace>::<name>`. All names that are not explicitly defined in a module are defined in the current module. If no modules have yet been defined, this is `MAIN`. If a name is explicitly referenced, then only that module will be searched. Otherwise, the current module will first be searched, and if no name is matched then `MAIN` will be searched.

```
(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it and
    assert the answer"
  (declare (auto-focus TRUE))
  (MAIN::question (ident ?id) (text ?text) (type
    ?type))
  (not (MAIN::answer (ident ?id)))
  ?ask <- (MAIN::ask ?id)
  =>
  (bind ?answer (ask-user ?text ?type))
  (assert (answer (ident ?id) (text ?answer)))
  (retract ?ask)
  (return))
```

Figure 4.1: An example of the use of `auto-focus` and `return`.

4.4 Control flow

Before we attempt to code an example application, we will still need a few control flow constructs that we can use in a function or in the right hand side of a rule. The following are equivalent to their counterparts in Java:

- `(foreach <var> <list> <statement>*)` calls the list of `<statement>`s once for each element in `<list>`, with that element bound to the variable `<var>`
- `(while <Boolean expression> <statement>*)` repeats the list of statements in `<statement>*` while `<Boolean expression>` evaluates to `TRUE`
- `(if <Boolean expression> then <statement>* else <statement'>*)` executes the list of `<statement>`s if `<Boolean expression>` evaluates to `TRUE`, and it evaluates the list of `<statement'>`s otherwise.

It is worth mentioning that, like specifying salience values, these constructs are not declarative: you are specifying the order in which things should happen. By using these constructs instead of declarative patterns, you are turning down the advantages of the built-in rule engine and conflict resolution and reasoning and writing your own algorithm. Depending on the task you are modelling, you may get a more readable program by just asserting facts and then letting the system fire other rules as a result, rather than defining the sequence of events yourself. Since most problems will require a combination of declarative and imperative reasoning, and since Jess allows combinations of imperative and declarative programming, you should make sure you are not inadvertently modelling a problem imperatively that would be better modelled declaratively (or vice versa!). As with the salience values, if you find yourself writing a program mainly in such imperative constructs, perhaps Jess is not the best language for the task; in fact in the second paper on Jess we will see that Jess provides capabilities for implementing functions in Java and importing them into a Jess session for precisely this situation.

In addition to these control flow statements, Jess provides a few more convenient statements:

- `(progn <statement>*)` executes the list of `<statement>`s, and returns the value of the last `<statement>` as if the whole list had been a single expression. This is useful when you want to use a list of expressions in a situation where only a single expression is allowed (such as the first argument to `(while)`).
- `(apply <variable> <argument>*)` treats the value of `<variable>` as the name of a function, and calls that function with the `<argument>`s.
- `(eval <string>)` and `(build <string>)` treat the value of `<string>` as Jess code and evaluate it.

Chapter 5

An example application

Now we have covered Jess sufficiently to implement a complete program. Earlier we mentioned as an example a "tax forms advisor" system, that interviews a user about his or her financial situation, and then recommends the appropriate USA tax forms for that user; we will build this system now. To improve readability, the examples in this text will be condensed versions of the code; however for completeness the entire code is reproduced in appendix A and occasionally the text will refer to the appropriate section of this appendix.

5.1 The knowledge base

The first step in developing an expert system is to collect all the knowledge we might need about the domain; in this case, this is the knowledge about which tax forms are applicable to which kinds of people. In practice, this step would involve a domain expert and significant research; for our purposes (we are just demonstrating the system!) we will assume we already have all this knowledge. We summarise the knowledge as follows:

- #1040 is the standard long form.
- #1040A is the short form (use instead of #1040 if income is less than $50,000 and there are no deductions)
- #1040EZ is the very short form (use instead of #1040A if income is less than $50,000 and there are no dependents, no deductions and no taxable interest above $400.)
- #2441 is for daycare expenses (including elderly parents and other dependents)
- #2016EZ is for unreimbursed work expenses (primarily travel)
- #3903 is for unreimbursed moving expenses if you moved (further than 50 miles) because of your job

- #4684 is for recovering losses
- #4868 is for applying for an extension for filling out taxes
- #8283 is for credit for donating more than $500 to charity
- #8829 is for deducting home office expenses

5.2 The facts

Each of these points becomes a Jess rule, for example:

```
(defrule form-1040EZ
  "1040EZ is the very short form (use instead of \#1040A if income is less than
          \$50,000 and there are no dependents, no deductions and no taxable
          interest above \$400.)"
  (user (income ?i&:(< ?i 50000))
        (dependents ?d&:(eq ?d 0)))
  (answer (ident interest) (text no))
  =>
  (assert (recommendation
           (form 1040EZ)
           (explanation "Income below threshold, no dependents"))))

(defrule form-1040A
  "1040A is the short form (use instead of \#1040 if income is less than
          \$50,000 and there are no deductions)"
  (user (income ?i&:(< ?i 50000))
        (dependents ?d&:(> ?d 0)))
  =>
  (assert (recommendation
           (form 1040A)
           (explanation "Income below threshold, with dependents"))))
```

These are contained in the module `recommend` (section A.5). And to be honest, that's most of the work done.

5.3 Our infrastructure

As well as the facts we have just enumerated, we will need some "infrastructure" to interact with the system. The typical run of the program will run as follows:

1. Welcome the user
2. Ask questions of the user
3. Decide which forms are appropriate

4. Notify the user of the appropriate forms

These four steps suggest a natural division into four modules:

1. `startup`,
2. `interview`,
3. `recommend`, and
4. `report`.

In addition to these modules, however, we will need a rule that says, basically, "if we have a question but no answer, ask the question". This rule cannot really live in any of these modules, since we always need the answers to our questions to operate. Hence we will create a new module to hold this rule - the `ask` module - and give this rule `auto-focus`:

```
(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it and assert the answer"
  (declare (auto-focus TRUE))
  (MAIN::question (ident ?id) (text ?text) (type ?type))
  (not (MAIN::answer (ident ?id)))
  ?ask <- (MAIN::ask ?id)
  =>
  (bind ?answer (ask-user ?text ?type))
  (assert (answer (ident ?id) (text ?answer)))
  (retract ?ask)
  (return))
```

Since all our facts (questions, user answers, and generated recommendations) are required by multiple modules, they will all live in `MAIN`.

To welcome a user, we will merely print out a friendly welcome message; we can do this with a rule that has no left hand side, and `(printout)` functions on its right hand side:

```
(defrule print-banner
  =>
  (printout t "Type your name and press Enter> ")
  (bind ?name (read))
  (printout t crlf "**********************************" crlf)
  (printout t " Hello, " ?name "." crlf)
  (printout t " Welcome to the tax forms advisor" crlf)
  (printout t " Please answer the questions and" crlf)
  (printout t " I will tell you what tax forms" crlf)
  (printout t " you may need to file." crlf)
  (printout t "**********************************" crlf crlf))
```

The recommendation module consists of all the rules we previously defined (our knowledge), plus one rule that concatenates different explanations for the same form, if both explanations are in our fact base:

```
(defrule combine-recommendations
  ?r1 <- (recommendation (form ?f) (explanation ?e1))
  ?r2 <- (recommendation (form ?f) (explanation ?e2&:(neq ?e1 ?e2)))
  =>
  (retract ?r2)
  (modify ?r1 (explanation (str-cat ?e1 ?*crlf* ?e2))))
```

The output module has rules that nicely print all the `recommendation` facts in our knowledge base (see section A.6):

```
(defrule sort-and-print
  ?r1 <- (recommendation (form ?f1) (explanation ?e))
  (not (recommendation (form ?f2&:(< (str-compare ?f2 ?f1) 0))))
  =>
  (printout t "*** Please take a copy of form " ?f1 crlf)
  (printout t "Explanation: "  ?e crlf crlf)
  (retract ?r1))
```

That just leaves the `interview` and `ask` modules.

5.4 Questions and the `ask` module

The questions are defined as `question` facts, with slots for a unique identifier, expected answer type, and the question text:

```
(deffacts question-data
  "The questions the system can ask."
  (question (ident income) (type number)
            (text "What was your annual income?"))
  (question (ident interest) (type yes-no)
            (text "Did you earn more than $400 of taxable interest?"))
  (question (ident dependents) (type number)
            (text "How many dependents live with you?"))
    . . .
  (question (ident home-office) (type yes-no)
            (text "Did you work in a home office?")))
```

This way the `ask` module can do some basic type checking on your answer (yes/no vs numeric, for example). These facts are set to be active at system startup with the `(deffacts)` function (see section A.1). The `ask-question-by-id` rule (section A.2) has already been seen in figure 4.1; this asks a question as soon as an `ask` fact appears in the fact base:

```
(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it and
    assert the answer"
  (declare (auto-focus TRUE))
  (MAIN::question (ident ?id) (text ?text) (type
    ?type))
  (not (MAIN::answer (ident ?id)))
  ?ask <- (MAIN::ask ?id)
  =>
  (bind ?answer (ask-user ?text ?type))
  (assert (answer (ident ?id) (text ?answer)))
  (retract ?ask)
  (return))
```

5.5 The `interview` module

The `interview` module (section A.4) is responsible for determining which questions need to be asked, and asserting the necessary `ask` facts. Most of the rules in the `interview` module have no LHS, so they can always be fired, for example:

```
(defrule request-income
  =>
  (assert (ask income)))
```

However, some of the questions depend on the answers to other questions, so we have some rules that only ask a question if conditions are satisfied:

```
(defrule request-childcare-expenses
  ;; If the user has dependents
  (answer (ident dependents) (text ?t&:(> ?t 0)))
  =>
  (assert (ask childcare)))
```

The `interview` module also contains one special rule `assert-user` that asserts a `user` fact once we know the user's income and number of dependents:

```
(defrule assert-user-fact
  (answer (ident income) (text ?i))
  (answer (ident dependents) (text ?d))
  =>
  (assert (user (income ?i) (dependents ?d))))
```

5.6 Running the system

Now we have all the components of the system; we merely need to run it. This is done by a function `(run-system)` (section A.7) that resets the working memory, initialises the focus stack, and then starts the program with `(run)`:

```
(deffunction run-system ()
  (reset)
  (focus startup interview recommend report)
  (run))
```

To run this system continuously, as a kiosk for example, the final command in the Jess file repeatedly calls this function inside a `(while TRUE)` loop:

```
(while TRUE
  (run-system))
```

Chapter 6

Conclusion

We have introduced Jess, the Java Expert System Shell. Jess is both a rule-based language for specifying expert systems, and a rule-based system for executing expert systems written in the Jess language. An expert system is a system that can reason about facts about the world using rules, and take appropriate actions as a result.

The Jess system is implemented in Java, and can be run both as a stand-alone interpreter (interpreting individual Jess strings, or whole files containing Jess code) or from Java code. Jess is also able to call external Java code.

The Jess language has a Lisp-like syntax, similar to its predecessor rule-based system CLIPS. We outlined the basics of the Jess language: symbols, values, variables, lists, functions, conditional elements, facts, rules and constructs. We also saw a number of useful built-in functions and constructs, both for debugging and for list, variable and fact manipulation.

In chapter 4 we introduced the execution cycle of Jess, as well as mechanisms provided by Jess to influence factors such as the order in which rules are fired, modularisation of rules and facts, and movement of focus between modules.

After this introduction to the Jess language and interpreter, we introduced a simple expert system, the Tax Form Advisor, that is able to recommend tax forms to a user after asking the user a number of questions about his or her financial and household situation. We implemented this expert system in Jess using the features introduced in this paper.

Bibliography

[Hill(2003)] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.

[Hill(2006)] Ernest Friedman Hill. Jess, the rule engine for the java platform, 2006. URL `http://herzberg.ca.sandia.gov/jess/docs/70/`.

Appendix A

The Tax Forms Advisor source code

This source code is taken from [Hill(2003)]

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Income tax forms advisor from part 3 of "Jess in Action"
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

A.1 Module MAIN

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Module MAIN

(deftemplate user
  (slot income (default 0))
  (slot dependents (default 0)))

(deftemplate question
  (slot text)
  (slot type)
  (slot ident))

(deftemplate answer
  (slot ident)
  (slot text))

(deftemplate recommendation
  (slot form)
  (slot explanation))
```

```
(deffacts question-data
  "The questions the system can ask."
  (question (ident income) (type number)
            (text "What was your annual income?"))
  (question (ident interest) (type yes-no)
            (text "Did you earn more than $400 of taxable interest?"))
  (question (ident dependents) (type number)
            (text "How many dependents live with you?"))
  (question (ident childcare) (type yes-no)
            (text "Did you have dependent care expenses?"))
  (question (ident moving) (type yes-no)
            (text "Did you move for job-related reasons?"))
  (question (ident employee) (type yes-no)
            (text "Did you have unreimbursed employee expenses?"))
  (question (ident reimbursed) (type yes-no)
            (text "Did you have reimbursed employee expenses, too?"))
  (question (ident casualty) (type yes-no)
            (text "Did you have losses from a theft or an accident?"))
  (question (ident on-time) (type yes-no)
            (text "Will you be able to file on time?"))
  (question (ident charity) (type yes-no)
            (text "Did you give more than $500 in property to charity?"))
  (question (ident home-office) (type yes-no)
            (text "Did you work in a home office?")))

(defglobal ?*crlf* = "
")
```

A.2 Module ask

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Module ask

(defmodule ask)

(deffunction ask-user (?question ?type)
  "Ask a question, and return the answer"
  (bind ?answer "")
  (while (not (is-of-type ?answer ?type)) do
         (printout t ?question " ")
         (if (eq ?type yes-no) then
           (printout t "(yes or no) "))
         (bind ?answer (read)))
  (return ?answer))
```

```
(deffunction is-of-type (?answer ?type)
  "Check that the answer has the right form"
  (if (eq ?type yes-no) then
    (return (or (eq ?answer yes) (eq ?answer no)))
    else (if (eq ?type number) then
           (return (numberp ?answer)))
    else (return (> (str-length ?answer) 0))))

(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it and assert the answer"
  (declare (auto-focus TRUE))
  (MAIN::question (ident ?id) (text ?text) (type ?type))
  (not (MAIN::answer (ident ?id)))
  ?ask <- (MAIN::ask ?id)
  =>
  (bind ?answer (ask-user ?text ?type))
  (assert (answer (ident ?id) (text ?answer)))
  (retract ?ask)
  (return))
```

A.3 Module startup

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Module startup

(defmodule startup)

(defrule print-banner
  =>
  (printout t "Type your name and press Enter> ")
  (bind ?name (read))
  (printout t crlf "**********************************" crlf)
  (printout t " Hello, " ?name "." crlf)
  (printout t " Welcome to the tax forms advisor" crlf)
  (printout t " Please answer the questions and" crlf)
  (printout t " I will tell you what tax forms" crlf)
  (printout t " you may need to file." crlf)
  (printout t "**********************************" crlf crlf))
```

A.4 Module interview

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Module interview
(defmodule interview)
```

```
(defrule request-income
  =>
  (assert (ask income)))

(defrule request-num-dependents
  =>
  (assert (ask dependents)))

(defrule assert-user-fact
  (answer (ident income) (text ?i))
  (answer (ident dependents) (text ?d))
  =>
  (assert (user (income ?i) (dependents ?d))))

(defrule request-interest-income
  ;; If the total income is less than 50000
  (answer (ident income) (text ?i&:(< ?i 50000)))
  ;; .. and there are no dependents
  (answer (ident dependents) (text ?d&:(eq ?d 0)))
  =>
  (assert (MAIN::ask interest)))

(defrule request-childcare-expenses
  ;; If the user has dependents
  (answer (ident dependents) (text ?t&:(> ?t 0)))
  =>
  (assert (ask childcare)))

(defrule request-employee-expenses
  =>
  (assert (ask employee)))

(defrule request-reimbursed-expenses
  ;; If there were unreimbursed employee expenses...
  (answer (ident employee) (text ?t&:(eq ?t yes)))
  =>
  (assert (ask reimbursed)))

(defrule request-moving
  =>
  (assert (ask moving)))

(defrule request-casualty
  =>
  (assert (ask casualty)))
```

```
(defrule request-on-time
  =>
  (assert (ask on-time)))

(defrule request-charity
  =>
  (assert (ask charity)))

(defrule request-home-office
  =>
  (assert (ask home-office)))
```

A.5 Module `recommend`

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Module recommend
(defmodule recommend)

(defrule combine-recommendations
  ?r1 <- (recommendation (form ?f) (explanation ?e1))
  ?r2 <- (recommendation (form ?f) (explanation ?e2&:(neq ?e1 ?e2)))
  =>
  (retract ?r2)
  (modify ?r1 (explanation (str-cat ?e1 ?*crlf* ?e2))))

(defrule form-1040EZ
  (user (income ?i&:(< ?i 50000))
        (dependents ?d&:(eq ?d 0)))
  (answer (ident interest) (text no))
  =>
  (assert (recommendation
           (form 1040EZ)
           (explanation "Income below threshold, no dependents"))))

(defrule form-1040A-excess-interest
  (user (income ?i&:(< ?i 50000)))
  (answer (ident interest) (text yes))
  =>
  (assert (recommendation
           (form 1040A)
           (explanation "Excess interest income"))))

(defrule form-1040A
  (user (income ?i&:(< ?i 50000))
        (dependents ?d&:(> ?d 0)))
  =>
  (assert (recommendation
           (form 1040A)
           (explanation "Income below threshold, with dependents"))))

(defrule form-1040-income-above-threshold
  (user (income ?i&:(>= ?i 50000)))
  =>
  (assert (recommendation
           (form 1040)
           (explanation "Income above threshold"))))

(defrule form-2441
  (answer (ident childcare) (text yes))
  =>
  (assert (recommendation
           (form 2441)
           (explanation "Child care expenses"))))

(defrule form-2016EZ
  (answer (ident employee) (text yes))
  (answer (ident reimbursed) (text no))
  =>
  (bind ?expl "Unreimbursed employee expenses")
  (assert
   (recommendation (form 2016EZ) (explanation ?expl))
   (recommendation (form 1040) (explanation ?expl))))

(defrule form-2016
  (answer (ident employee) (text yes))
  (answer (ident reimbursed) (text yes))
  =>
  (bind ?expl "Reimbursed employee expenses")
  (assert
   (recommendation (form 2016) (explanation ?expl))
   (recommendation (form 1040) (explanation ?expl))))

(defrule form-3903
  (answer (ident moving) (text yes))
  =>
  (bind ?expl "Moving expenses")
  (assert
   (recommendation (form 3903) (explanation ?expl))
   (recommendation (form 1040) (explanation ?expl))))
```

```
(defrule form-4684
  (answer (ident casualty) (text yes))
  =>
  (bind ?expl "Losses due to casualty or theft")
  (assert
   (recommendation (form 4684) (explanation ?expl))
   (recommendation (form 1040) (explanation ?expl))))

(defrule form-4868
  (answer (ident on-time) (text no))
  =>
  (assert
   (recommendation (form 4868) (explanation "Filing extension"))))

(defrule form-8283
  (answer (ident charity) (text yes))
  =>
  (bind ?expl "Excess charitable contributions")
  (assert
   (recommendation (form 8283) (explanation ?expl))
   (recommendation (form 1040) (explanation ?expl))))

(defrule form-8829
  (answer (ident home-office) (text yes))
  =>
  (bind ?expl "Home office expenses")
  (assert
   (recommendation (form 8829) (explanation ?expl))
   (recommendation (form 1040) (explanation ?expl))))
```

A.6 Module report

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Module report

(defmodule report)

(defrule sort-and-print
  ?r1 <- (recommendation (form ?f1) (explanation ?e))
  (not (recommendation (form ?f2&:(< (str-compare ?f2 ?f1) 0))))
  =>
  (printout t "*** Please take a copy of form " ?f1 crlf)
  (printout t "Explanation: "  ?e crlf crlf)
  (retract ?r1))
```

A.7 Control commands

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Test data

(deffunction run-system ()
  (reset)
  (focus startup interview recommend report)
  (run))

(while TRUE
  (run-system))
```