The Working Memory of an Expert System

The collection of facts in an expert system is known as the working memory.

- All the data that rules operate on, are represented as facts in the working memory.

Example: An expert system which is an environmental control system of an office building.

The facts could be the temperature and humidity readings from around the building and sensor readings and switch settings from the building’s air-conditioning system.

(temperature floor1 20.5)
(temperature floor2 22.0)
(humidity floor1 55)
(humidity floor2 61)

Jess functions related to facts

- **assert**: Add facts to working memory.
- **clear**: Clears Jess. Deletes facts, rules, functions, etc.
- **deffacts**: Defines the initial contents of working memory.
- **facts**: Displays the contents of working memory.
- **reset**: Initialises the working memory. Removes all facts, inserts the fact **(initial-fact)**, then it insert all facts found in **deffacts**.
- **retract**: Removes facts from working memory.
- **watch**: Prints diagnostics when events happen.

The watch function

Print messages when various interesting events happen.

For example, with **watch facts** a message appears when an event is added or removed.

Jess> (watch facts)
TRUE
Jess> (reset)
=> f-0 (MAIN::initial-fact)
TRUE
Jess> (unwatch facts)
TRUE
Jess> (reset)
TRUE
The facts function
Lists all the facts in the working memory.

Example:
Jess> (facts)
f-0  (MAIN::initial-fact)
For a total of 1 facts.

Creating and removing facts
- The assert function inserts facts into the working memory.
- The retract function removes facts from the working memory.

Example:
(reset)
TRUE
Jess> (assert (groceries milk eggs bread))
<Fact-1>
Jess> (facts)
f-0  (MAIN::initial-fact)
f-1  (MAIN::groceries milk eggs bread)
For a total of 2 facts.
Jess> (retract 1)
TRUE
Jess> (facts)
f-0  (MAIN::initial-fact)
For a total of 1 facts.

Clearing and initialising working memory
- The reset function restores the initial state of an application without erasing it.
- The initial-fact which is inserted after reset, is used internally by Jess.
- The initial contents (facts) of the working memory (in addition to initial-fact) can be specified with the deffacts function.
Example:

Jess> (clear)
TRUE
Jess> (deffacts catalog "Product-catalog"
   (product 354 pc-athlon-64 "$399"
   (product 355 pc-pentium-4 "$419"
   (product 356 laser-printer "$120"
   (product 357 colour-printer "$139")))
TRUE
Jess> (reset)
TRUE
Jess> (facts)
TRUE
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::product 354 pc-athlon-64 "$399")
f-2 (MAIN::product 355 pc-pentium-4 "$419")
f-3 (MAIN::product 356 laser-printer "$120")
f-4 (MAIN::product 357 colour-printer "$139")
For a total of 5 facts.

Representation of facts in Jess

The implementation of working memory in Jess, is similar to a relational database, with facts corresponding to the rows of a database.

- Good performance can be achieved through appropriate indexing.
  To be able to do that, Jess supports three types of facts:
  1. Unordered facts: columns are specified in any order:
     (person (name "richard") (age 22) (height 1.80) (weight 75))
  2. Ordered facts: just a flat list in which the position of an element indicates the corresponding column.
     (person "richard" 22 1.80 75)

Unordered Facts

Before using an unordered fact, its structure must be defined using the deftemplate function.

- The structure of an unordered fact defines the fields (slots) that a fact can contain.

Example:

Jess> (deftemplate person "people in company"
   (slot name)
   (slot age)
   (slot gender))
TRUE
Jess> (assert (person (age 34) (name "Bob") (gender male)))
<Fact-0>
Jess> (assert (person (age 20) (name "Helen")))
<Fact-1>
For a total of 2 facts.

To specify default values:

Jess> (deftemplate person "people in company"
   (slot name) (slot age) (slot gender (default male)))
TRUE
Jess> (assert (person (age 20) (name "Helen") (gender female)))
<Fact-0>
Jess> (assert (person (age 34) (name "Bob")))
<Fact-1>
Jess> (facts)
f-0 (MAIN::person (name "Bob") (age 34) (gender male))
f-1 (MAIN::person (name "Helen") (age 20) (gender nil))
For a total of 2 facts.

To a total of 2 facts.
Multislots in unordered facts
Unlike “normal” slots, multislots of an unordered fact can hold multiple values.

Example:
(deftemplate person "people in company"
  (slot name)
  (slot age)
  (slot gender (default male))
  (multislot hobbies))

(assert (person (age 18)
  (name "George")
  (hobbies tennis "swimming" football)))

Changing the slot values of existing facts
Function modify is used to change the values of a fact in the working memory.

Jess> (facts)
f-0 (MAIN::person (name "George") (age 18) (gender male)
  (hobbies tennis "swimming" football))
For a total of 1 facts.

Jess> (modify 0 (age 20))
<Fact-0>
Jess> (facts)
f-0 (MAIN::person (name "George") (age 20) (gender male)
  (hobbies tennis "swimming" football))
For a total of 1 facts.

Writing rules in Jess
The knowledge base of an expert system is a set of rules operating on facts.

- The defrule function is used to define a rule.
- A rules resembles an if ... then statement. The then part can be executed (i.e. the rule fires), when the if part is satisfied by the existing facts in the memory.
- Jess decides the order in which to fire the rules.

Example:
The simplest rule does not have a condition and it has a single action which prints a message:

Jess> (defrule simple-rule
  "A rule that prints a message"
  =>
  (printout t "simple rule: hello" crlf))
TRUE
Jess> (watch facts)
TRUE
Jess> (watch activations)
TRUE
Jess> (watch rules)
TRUE
In the previous example, the following events take place:

1. The function call (watch activations) instructs Jess to print a message whenever an activated rule is placed on, or removed from the agenda.

2. The reset call, inserts the fact initial-fact in memory.

3. Rules without a condition are triggered by initial-fact, i.e. simple-rule is placed in the agenda.

4. After run is issued, Jess’s rule engine fires the rules on the agenda, one at a time, after placing them in the execution engine, until the agenda is empty.

5. run returns the number of rules fired, e.g. 1 in the above example.

Other functions related to rules

- ppdefrule: Pretty-prints a rule.
- run: Begins firing activated rules from the agenda. It returns the number of rules fired.
- undefrule: Deletes a rule.
- watch rules: Prints a diagnostic when a rule fires.

An example of a more complex rule

(defrule drink-if-thirsty
  "If thirsty, then drink something"
  ?thirst <- (is-thirsty)
  =>
  (printout t "What would you like to drink? ")
  (bind ?drink (read))
  (have-a-drink ?drink)
  (retract ?thirst))
A complete program

(clear)

(watch all)

(reset)

(defrule drink-if-thirsty
  "If thirsty, then drink something"
  ?thirst <- (is-thirsty)
  =>
  (printout t "What would you like to drink? ")
  (bind ?drink (read))
  (have-a-drink ?drink)
  (retract ?thirst))

(deffunction have-a-drink (?drink)
  (if (eq ?drink water) then
      (printout t "Not thirsty any more!" crlf)
    else
      (printout t "This drink is not good for your health" crlf)))

(assert (is-thirsty))

(run)

Specifying Variables as Constraints

Jess> (defrule repeated-variables
  (a ?x)
  (b ?x)
  =>
  (printout t "?x is " ?x crlf))

TRUE

Jess> (watch activations)

TRUE

Jess> (deffacts repeated-variables-facts
  (a 1)
  (a 2)
  (b 2)
  (b 3))

TRUE

Jess> (reset)
Multifields in Rules
Multifields can match any number of values (zero or more).

- Multifields can be only used in multislots.

(deftemplate shopping-cart "shopping cart"
  (multislot contents))

(defrule any-shopping-cart
  (shopping-cart (contents $?items))
  =>
  (printout t "The cart contains " ?items crlf))

Note that the dollar sign can be left out in the right hand side of a rule.

Connective constraints
Match with not exact patterns. For example, match a client if located either in Boston or Hartford.

- The connective constraints are: & (and), l (or) and ~ (not).

Examples:
- city is either Boston or Hartford:
  (client (city Boston|Hartford))
- city of client is not Bangor:
  (client (city ?c&~Bangor))
- clients neither from Bangor nor from Portland:
  (client (city "Bangor&Portland"))
- purchased exactly two items which are not the same:
  (client (items-purchased ?x ~?x))

Constraining matches with predicate functions
Predicate functions can be used when connective constraints cannot express particular patterns.

- Predicate functions return TRUE or FALSE.
- A predicate constraint is specified using a predicate function preceded by a colon (:).

Examples:
- Print a message to any customers who go to the checkout with less than 5 items:
  (deftemplate shopping-cart ""
    (slot customer-id)
    (multislot contents))

(defrule small-order
  (shopping-cart (customer-id ?id)
    (contents $?c&:(< (length$ $?c) 5)))
  =>
  (printout t "Wouldn't you like to buy more?" crlf))

- A rule firing if a customer is checking out with more than 5 items, but his cart contains neither milk nor butter:
  (defrule large-order-and-no-dairy
    (shopping-cart (customer-id ?id)
      (contents $?c&:
        (and (> (length$ $?c) 5)
          (not (or (member$ milk $?c)
            (member$ butter $?c)))))
      (checking-out-now ?id)
      =>
      (printout t "Don't you need dairy products?" crlf))
Matching patterns between facts

To express relationships between different facts, conditional elements can be used.

- Although some of the conditional elements have the same names as predicate functions, unlike predicate function which operate on boolean expressions, conditional elements operate on patterns.
  - and: matches multiple facts.
  - or: matches alternative facts.
  - not: matches if no facts match.
  - exists: matches if at least one fact matches.
  - test: matches if a function call doesn't evaluate to FALSE.
  - logical: matching facts offer logical support to new facts.

Conditional Elements (and) - Example 1:
(defrule ready-to-fly
  (and (flaps-up)
       (engine-on))
  =>)

Conditional Elements (or) - Example 2:
(deftemplate used-car (slot price) (slot mileage))
(deftemplate new-car (slot price) (slot warrantyPeriod))
(defrule might-buy-car
  (?candidate <- (or (used-car (mileage ?m & : (< ?m 50000)))
                     (new-car (price ?p & : (< ?p 20000))))
    =>
    (assert (candidate ?candidate)))
(assert (new-car (price 18000)))
(assert (used-car (mileage 30000)))

Conditional Elements (not) - Example 3:
The not pattern matches the absence of a fact.
Fire a rule when no red cars are found:
(defrule no-red-cars
  (not (auto (colour red)))
  =>)
Contrast with the following which fires for every car that is not red:
(defrule no-red-cars
  (auto (colour ~red))
  =>)
Conditional Elements (exists) - Example 4:

- An exists conditional element is true if there exist any facts that match the pattern.
- The exists element will cause a rule to fire only once, even if there are many facts which activate it.

(defrule exists-an-honest-man
  (exists (honest ?))
  =>
  (printout t "There is at least one honest man." crlf))

Conditional Elements (test) - Example 5:

(deftemplate person (slot age))
(deftemplate person (slot age))

deftemplate person (slot age)
(defrule find-trustworthy-people1
  (person (age ?x))
  (test (< ?x 30))
  =>
  (printout t ?x " is under 30!" crlf))

Equivalent approach using predicate constraints:
(deftemplate person (slot age))
(defrule find-trustworthy-people2
  (person (age ?x& : (< ?x 30)))
  =>
  (printout t ?x " is under 30!" crlf))

Backward-chaining rules in Jess

- Goal seeking approach.
- The engine tries to make rules fire.
- If the if clause of one rule is only partially matched, and the engine can determine that firing another rule would cause it to be fully matched, it will try to fire the second rule.
- In Jess, the facts that serve as backward-chaining triggers must be declared explicitly.

deftemplate person (slot age)

deftemplate person (slot age)
(deftemplate person (slot age))

deftemplate person (slot age)

Example - Calculating the factorial

deftemplate person (slot age)

deftemplate person (slot age)

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

(defrule do-factorial
  (need-factorial ?x ?)
  =>
  ;; compute the factorial of ?x in ?r
  (bind ?r 1)
  (bind ?n ?x)
  (while (> ?n 1)
    (bind ?r (* ?r ?n))
    (bind ?n (- ?n 1)))
  (assert (factorial ?x ?r)))
Partitioning the rule base

The `defmodule` allows to divide groups and facts into groups called `modules`. The default module is called `MAIN`.

- The rules in a module can fire only when the module has the focus.
- Only one module can be in focus at a time.

Functions related to modules:

- `defmodule`: defines a new module.
- `focus`: set the focus module.
- `get-current-module`: returns the current module.

Example:

Jess> (defmodule WORK) TRUE
Jess> (deftemplate WORK::job (slot salary)) TRUE
Jess> (list-deftemplates WORK) WORK::job
For a total of 1 deftemplates.
Jess> (get-current-module) WORK
Jess> (defmodule COMMUTE) TRUE
Jess> (get-current-module) COMMUTE
Modules, Scope and Name Resolution

When Jess is compiling a rule, query or deffacts definition, it looks 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 default module MAIN is searched.

Example:

Jess> (clear) TRUE
Jess> (assert (MAIN::mortgage-payment 2000)) <Fact-0>
Jess> (deffacts)
TRUE
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))))
   (MAIN::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))"

The auto-focus declaration

When a rule that declares the auto-focus property is activated, its module automatically gets the focus.

Jess> (defmodule DRIVING)
Jess> (defmodule PROBLEMS)
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-0>
Jess> (assert (other-car 4))
<Fact-1>
Jess> (focus DRIVING)
MAIN
Jess> (run)
...Crash!
4