MESS -- A Minimal Expert System Shell

Larsson, Jan Eric

1988

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7380)/1-12/(1988)

MESS—A Minimal Expert System Shell

Jan Eric Larsson

Department of Automatic Control
Lund Institute of Technology
August 1988

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Technical report

Date of issue
August 1988

Document Number
CODEN: LUTFD2/(TFRT–7380)/1–12/(1988)

Author(s)
Jan Eric Larsson

Supervisor

Sponsoring organisation

Title and subtitle
MESS—A Minimal Expert System Shell

Abstract

This report describes a small expert system shell. The system is written in Chez Scheme and handles both forward and backward chaining. The source code is short, (192 lines), and the system is well suited for inclusion in larger programs. A small manual, the source code, and examples of rules and runs are included.

Key words
Chez Scheme, Expert System Shells, Production Systems

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
12

Recipient's notes

Security classification

MESS—A Minimal Expert System Shell

Jan Eric Larsson

Department of Automatic Control
Lund Institute of Technology
August 1988

Department of Automatic Control
Lund Institute of Technology
Box 118
S–221 00 LUND
Sweden

Published 1988
Printed in Sweden

Contents

Introduction

MESS is a small expert system shell which supports both forward and backward chaining. The rule syntax is simple and Lisp-like, but contains everything that is needed for a smaller project. There is a small number of utility functions and a simple tracing facility. The MESS system is written in Chez Scheme and the entire code is 192 lines long. The intention is that the user should be able to get a complete understanding of what is happening by reading the code. The system should also be easy to include in a larger program. For readings about Scheme, see Abelson, Sussman, and Sussman, [1985], and Dybvig [1987]. An inspiration for this project is found in Winston and Horn [1981], where a small rule-based system is outlined.

The Rule Syntax

A rule has the following syntax:

```
(rule <name>
  (if
    <pattern>
     ...
    <pattern>)
  (test
    <Lisp expression>
     ...
    <Lisp expression>)
  (then
    <pattern>
     ...
    <pattern>))
```

The name of the rule should be either an atom or a string. The `test` part is optional, but the `if` and `then` parts must exist. The patterns used in the `if` and `then` parts may be arbitrarily nested Lisp lists. These are not proper facts, but matching patterns. During execution, they will be matched against the facts in the database.

An atom in a pattern starting with a '-' is a variable and will match any expression in the corresponding fact. If the variable occurs several times in a rule, it must match the same expression everywhere. The variable '-' is the anonymous variable and will match different expressions in the same rule. The anonymous variable may also occur in the facts in the database. This can be especially useful for database queries in backward chaining, to allow for incomplete hypotheses.

Negation of a pattern is written as `(not <pattern>)`. A rule containing such a negation will only be triggered if there is no corresponding fact to match it in the database.

In backward chaining, the system may be allowed to ask for the truth of matching patterns. Such a matching pattern must start with a '*'. This '*' will be cdr'ed off, and is not used in matching with the facts.

The patterns in the `then` part may contain a '~'. This means that the next item will be evaluated before the pattern is used. Three special functions are also available in the `then` part. A pattern of the form `(remove <pattern>)` means that the corresponding fact should be removed from the database. A pattern of the form `(call <pattern >)` means that the `cdr` will be used as a function call. In both these cases, the bodies may contain variables. A patterns of the form `(ask <pattern>)` means that the pattern will be queried for.

The facts in the database and the hypotheses are also arbitrarily nested Lisp lists, but they may not contain any variables, except for the anonymous variable '-'.

The Database

The system keeps its data in three different lists. The rules should be put in a list named `rules`. The facts reside in a list called `facts`. The hypotheses to be tried in backward chaining should be put in the list `hypotheses`.

A simple tracing facility is available. If the variable `trace` is non-`nil`, a message will be printed every time a rule is used, in forward and backward chaining.

Forward Chaining

The forward chaining is started with the top-level function (`fc`). Some initial facts must be present in the database, i.e., the `facts` list. The forward chaining strategy matches the `if` parts of rule after rule against the facts in the database. When the entire `if` part of a rule matches, the tests of the `test` part are evaluated. The test expressions must all return non-`nil` values. In that case, the `then` part is applied. Patterns starting with `remove`, `call`, or `ask` have special effects. Other patterns will be added to the database if not already there. The rule will be successfully used only if at least one pattern is added to or deleted from the database. Once a rule has been found and successfully applied, the system starts all over again, searching for a rule that will match the updated database. The forward chaining is finished when no rule can be triggered.

The forward chaining uses a very simple conflict resolution strategy. The rules are searched from the beginning of the `rules` list, and the first rule to match will be chosen. In the matching of each pattern, the `facts` list is searched from the beginning. As new facts are cons'ed to the front of this list, more recent facts will be preferred over older ones. Every time a rule has been used, the system starts anew from the beginning of the `facts` list. This simple strategy is efficient and may easily be controlled by the order of the rules in the `rules` list.

Backward Chaining

The backward chaining is started with the top-level function (`bc`). Some facts may or may not be present in the database, and at least one fact should be placed in the `hypotheses` list. The system goes through the hypotheses, either until one is verified, or the list is empty. A hypothesis may be verified in several ways. If a corresponding fact is found in the database, the hypothesis is immediately verified. If it is found in the `then` part of a rule, the patterns of the `if` part become new hypotheses, and the system tries to verify them recursively. If a hypothesis starts with a '*', and can be verified neither with the database nor the rules, the system asks for its truth. The answer must be either of 'y' or 'n'. As a result of a query, a fact (`asked <pattern>`) is put into the database. Thus, the system will never ask for the same thing twice.

The backward chaining uses the same conflict resolution strategy as the forward chaining, i.e., the `rules` and `facts` lists are searched from the beginning. The search may thus be guided by the order of the rules in the list.

The Source Code

Here is the entire source code of the system. The global variables `rules`, `facts`, and `hypotheses` should be given the appropriate values.

```
;---MESS---A Minimal Expert System Shell-------------------------------------
;
;   Author: Jan Eric Larsson
;           Department of Automatic Control
;           Lund Institute of Technology, Lund, Sweden
;
;---Rule selectors-----------------------------------------------------------

(define (getifs rule)
  (cdr (assoc 'if (cddr rule))))

(define (getthens rule)
  (cdr (assoc 'then (cddr rule))))

(define (gettests rule)
  (if (null? (assoc 'test (cddr rule)))
    nil
    (cdr (assoc 'test (cddr rule)))))

;---Fact manipulation--------------------------------------------------------
```

```
(define (var? e)
  (and
    (symbol? e)
    (equal? (substring (symbol->string e) 0 1) "-")
    (not (equal? e '-))))

(define (fillin fact env)
  (cond
    ((null? fact) nil)
    ((and
      (var? (car fact))
      (assoc (car fact) env))
      (cons (cadr (assoc (car fact) env)) (fillin (cdr fact) env)))
    ((atom? (car fact)) (cons (car fact) (fillin (cdr fact) env)))
    (t (cons (fillin (car fact) env) (fillin (cdr fact) env)))))

(define (evalstuff fact)
  (cond
    ((null? fact) nil)
    ((equal? (car fact) '^) (cons (eval (cadr fact)) (evalstuff (cddr fact))))
    (t (cons (car fact) (evalstuff (cdr fact))))))

(define (addnewfacts fact env)
  (let ((f (evalstuff (fillin fact env))))
    (cond
      ((equal? (car f) 'remove)
        (if (member (cadr f) facts) (set! facts (remove! (cadr f) facts)) nil))
      ((equal? (car f) 'call) (eval (cadr f)) nil)
      ((equal? (car f) 'ask) (ask (cons '* (cadr f)) env))
      (t (set! facts (cons f facts))))))

;---Unification-----------------------------------------------------------

(define (unify p d env)
  (cond
    ((and (null? p) (null? d)) env)
    ((or (null? p) (null? d)) nil)
    ((or (equal? (car p) '-) (equal? (car d) '-)) (unify (cdr p) (cdr d) env))
    ((equal? (car p) '*) (unify (cdr p) d env))
    ((equal? (car p) (car d)) (unify (cdr p) (cdr d) env))
    ((not (var? (car p))) nil)
    ((null? (assoc (car p) env))
      (unify (cdr p) (cdr d) (cons (list (car p) (car d)) env)))
    ((equal? (cadr (assoc (car p) env)) (car d)) (unify (cdr p) (cdr d) env))
    (t nil)))

;---Forward chaining------------------------------------------------------

(define (getfact fact factlist env)
  (cond
    ((equal? (car fact) '*) (getfact (cdr fact) factlist env))
    ((null? factlist) (if (equal? (car fact) 'not) (list nil env) nil))
    ((equal? (car fact) 'not)
      (if (null? (getfact (cadr fact) factlist env))
        (list (cdr factlist) env)
        nil))
```

```
    ((let ((newe (unify fact (car factlist) env)))
       (if newe (list (cdr factlist) newe) nil)))
    (t (getfact fact (cdr factlist) env))))

(define (putfact fact factlist env)
  (cond
    ((null? factlist) (addnewfacts fact env))
    ((unify fact (car factlist) env) nil)
    (t (putfact fact (cdr factlist) env))))

(define (testif ifs rule env)
  (if ifs
    (do ((e (list facts env)) (newe t) (retval nil))
      ((or (null? newe) (equal? newe '(() ((((((  retval) retval)
      (set! newe (getfact (car ifs) (car e) (cadr e)))
      (if newe
        (begin
          (set! e (list (car newe) (cadr e)))
          (set! retval (testif (cdr ifs) rule (cadr newe))))))
    (checktest rule env)))

(define (checktest rule env)
  (do
    ((tests (gettests rule) (cdr tests)))
    ((or (null? tests) (not (eval (fillin (car tests) env))))
      (if tests nil (usethen rule env)))))

(define (usethen rule env)
  (do
    ((thens (getthens rule) (cdr thens)) (retval nil))
    ((null? thens) (if retval (list nil retval) nil))
    (if (putfact (car thens) facts env) (set! retval env))))

(define (findrule)
  (do
    ((rlist rules (cdr rlist)))
    ((or (null? rlist) (testif (getifs (car rlist)) (car rlist) '(((((  ))))))
      (if (and trace rlist)
        (begin
          (display "Rule ")
          (display (cadar rlist))
          (display " triggered")
          (newline)))
      (if (null? rlist) nil t))))

(define (fc) (if (findrule) (fc)) 'Ok)

;---Backward chaining-------------------------------------------------------

(define (verifyif ifs rule env)
  (if ifs
    (let ((e (verify (car ifs) facts env)))
      (if e (verifyif (cdr ifs) rule (cadr e))))
    (checktest rule env)))

(define (inthens? fact thens env)
```

```
  (cond
    ((null? thens) nil)
    ((unify (car thens) fact env))
    (t (inthens? fact (cdr thens) env))))

(define (userule fact env)
  (do
    ((rlist rules (cdr rlist)) (newe nil))
    ((or (null? rlist) newe) newe)
    (set! newe (inthens? fact (getthens (car rlist)) env))
    (if newe (set! newe (verifyif (getifs (car rlist)) (car rlist) newe)))
    (if (and trace newe)
      (begin
        (display "Rule ")
        (display (cadar rlist))
        (display " triggered")
        (newline)))))

(define (ask fact env)
  (cond
    ((and
      (equal? (car fact) '*)
      (null? (getfact (list 'asked (cdr fact)) facts env)))
      (set! facts (cons (list 'asked (cdr fact)) facts))
      (let ((f (cdr (fillin fact env))))
        (display "Is this true: ")
        (display (if (equal? (car f) 'not) (cadr f) f))
        (newline)
        (if (equal? (read) 'y)
          (if (equal? (car f) 'not)
            (begin (set! facts (cons (cadr f) facts)) nil)
            (begin (set! facts (cons f facts)) (list facts env)))
          (if (equal? (car f) 'not)
            (begin (set! facts (cons f facts)) (list facts env))
            (begin (set! facts (cons (list 'not f) facts)) nil)))))
    (t nil)))

(define (verify fact factlist env)
  (cond
    ((getfact fact factlist env))
    ((userule fact env))
    ((ask fact env))))

(define (bc)
  (do
    ((h hypotheses (cdr h)))
    ((or (null? h) (verify (car h) facts '(()))))
      (if (null? h)
        (display "No hypothesis verified")
        (begin (display (car h)) (display " verified")))
      (newline)))
  'Ok)

;---Global variables----------------------------------------------------------

(define trace nil)
```

```
(define rules nil)
(define facts nil)
(define hypotheses nil)
```

An Example of a Rule Base

This small example shows what a rule base might look like. The expertise in this case concerns how to "hang around in bars."

```
(define rules '(

(rule bar-1
  (if
    (bar open)
    (capital is -)
    (not (has a beer)))
  (then
    (buy a beer)))

(rule bar-2
  (if
    (buy a beer)
    (capital is -x))
  (test
    (> -x 9))
  (then
    (remove (buy a beer))
    (remove (capital is -x))
    (capital is ^ (- -x 10))
    (has a beer)))

(rule bar-3
  (if
    (has a beer))
  (then
    (call (display "Drink a beer"))
    (call (newline))
    (remove (has a beer))))))

(define facts '((bar open) (capital is 50)))

(define hypotheses '((buy a beer)))
```

Example Runs

This is what happens when the system is run with the rules and facts shown in the example above.

```
% scheme
Chez Scheme Version 2.0.1 Copyright (c) 1987 R. Kent Dybvig

> (load "mess.scm")
()
> (define trace t)
trace
> facts
((bar open) (capital is 50))
> (fc)
Rule bar-1 triggered
```

```
Rule bar-2 triggered
Drink a beer
Rule bar-3 triggered
Rule bar-1 triggered
Rule bar-2 triggered
Drink a beer
Rule bar-3 triggered
Rule bar-1 triggered
Rule bar-2 triggered
Drink a beer
Rule bar-3 triggered
Rule bar-1 triggered
Rule bar-2 triggered
Drink a beer
Rule bar-3 triggered
Rule bar-1 triggered
Rule bar-2 triggered
Drink a beer
Rule bar-3 triggered
Rule bar-1 triggered
ok
> facts
((buy a beer) (capital is 0) (bar open))
> (set! facts (cdr facts))
((capital is 0) (bar open))
> hypotheses
((buy a beer))
> (bc)
Rule bar-1 triggered
(buy a beer) verified
ok
> facts
((buy a beer) (capital is 0) (bar open))
> (exit)
%
```

References

Abelson, H., G. J. Sussman and J. Sussman, (1985): *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Massachusetts.

Dybvig, R. K., (1987): *The SCHEME Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey.

Winston, P. H. and B. K. P. Horn, (1981): *Lisp*, Addison-Wesley, Reading, Massachusetts.