A short introduction to OPS5

Peter Ross

This version: October 29, 1987

1 The language

This document contains a short guide to using OPS5. The Common LISP and Franz LISP versions are essentially the same. A short example program is included – the example is mainly intended to show the syntax and style, it is not meant to be a convincing argument in favour of the language.

1.1 Some useful commands

load OPS5 files as though they were normal LISP files, for example:

(load "filename") to load `filename.lsp` in Common LISP

(load 'filename) to load `filename.l` in Franz LISP

A word of warning to users of any Common LISP version of OPS5: there are several functions in OPS5 which happen to have the same name as standard Common LISP functions, in particular `remove`, `write` and `call`. There should be no problem if you use these on the right-hand side of an OPS5 rule, the OPS5 system should automagically rewrite them to some system dependent substitute. However, OPS5 allows you to use these at top level too, in which case you will need to find out what the substitute is. For example, the OPS5 `remove` might actually be implemented as `oremove`.

Here is a subset of the OPS5 top level commands. Some can be used in rules too:

(run)
: start the interpreter. You can also use the form (run *N*) if you want it to run for only *N* cycles.

(back *N*)
: run the interpreter backwards for *N* cycles.

(watch *N*)
: If the argument is 0, tracing is turned off. An argument of 1, 2 or 3 sets the trace level; 3 is the most detailed.

(remove *)
: deletes all of working memory. You can specify a sequence of time tags (see below) instead.

(make *pattern*)
: creates a working memory element.

(ppwm *pattern*)
: prints all working memory elements matching the pattern. If you give no pattern, all elements are printed. The related '`wm`' command expects one or more time tags rather than a pattern.

(pm *production names*)
: prints the named production rules.

(excise *production-names*)
: deletes the named production rules.

(pbreak *production-names*)
: sets or removes breakpoints on the named rules.

(matches *production-names*)
: prints information about partial matches for each named rule.

There are two conflict resolution strategies available in OPS5. Both are based on the idea of *recency*. Every working memory element has a *time tag*, just an integer, indicating "when" it was created or last modified. For any single production rule, there may be zero, one or more instances of that production rule in the set of possible rules to fire. There could be more than one instance if there happens to be more than one way in which the left-hand side conditions match working memory elements. OPS5 decides which of the whole set of instances of rules to fire, by the following process:

1. any instance which has already fired on an earlier cycle is discarded. For example, the rule instance which fired on the previous cycle may not have invalidated any of the working memory elements which matched the conditions of its left-hand side.

2. all remaining instances are ranked, according to a strategy-specific recency test. If there is a clear winner, it is the chosen one.

3. if there is no clear winner, consider the set of winners in the ranking test. Choose the one with most tests (for constants and variables) on its left-hand side. If there is more than one such, pick one at random.

In the LEX strategy, any two instances are first compared using the time tags of the most recent working memory elements of the two instantiations. If these are equal, the next most recent are compared, and so on. In a tie, the longer wins. The MEA (an acronym for "means-end analysis") strategy is almost the same; the difference is that the very first step is to compare the time tags of the two working memory elements which match the first condition elements of the two rule instantiations. So in MEA the first condition of a rule is somehow special; in a typical application, it might be used to denote 'the current goal'.

(strategy *type*)
sets which of the two available strategies to use.

1.2 Working memory elements

There are two types. The simpler is a plain sequence, of atoms and/or numbers, which looks just like a LISP list. For example:

```
(drink 2 more cups today)
```

The more useful is a named, unordered collection of attribute-value pairs. It also looks like a list. The name comes first, followed by the attribute-value pairs; the attribute name is marked by an initial up-arrow. For example:

```
(house ^type family ^age 12 ^rooms 6 ^walls stone ^roof wood)
```

Every attribute-value collection must be declared at the start of the program, like so:

```
(literalize house type age rooms walls roof)
```

The declaration should give the name and the attribute names. One of the attributes can be declared to be a vector-attribute if you want, in which case it can have an ordered sequence of values:

```
(vector-attribute children cars)
(literalize family husband wife children)
.....
  (family ^children john jim jane ^wife janet)
....
```

The vector-attribute declaration is global in scope.

It should be clear that, since these two types of working memory element are not recursively defined, it is difficult to represent any of the more elaborate data structures commonly used in other langauges. But it is possible to call arbitrary LISP functions from rules Internally, every working memory element has a time tag too.

1.3 Production rules

An OPS5 production rule looks like this:

```
(p rule-name
      patterns-describing-working-memory-elements
      -->
      actions )
```

Spaces, tabs and newlines can be used as you like to make the rules more readable. A pattern looks just like a LISP list, although it can be preceded by a minus sign to indicate that the pattern should **not** match **any** working memory element. The first pattern cannot be preceded by a minus sign.

1.3.1 The patterns on the left-hand side of a rule

A pattern begins with the name of a working memory element. If that is a plain sequence rather than an attribute-value type, then the pattern name must be followed by a sequence of *terms* (see below) which must match the parts of the sequence, in the correct order. If the named working memory element is an attribute-value collection, then the pattern name should be followed by pairs consisting of an attribute name (marked, as before, by an up-arrow) and a term which matches the corresponding value.

Terms can be any of the following:

constants just an atom or a number

variables indicated by an atom beginning with < and ending with >, such as <x> or <whisky>. The scope of a variable is the production rule in which it appears; it can match any atom or number. Note that the first occurrence of a variable in a rule must establish the binding for it, so you cannot start a rule with a pattern such as (size > <s>) (see the explanation of the > operator below).

disjunctions These start with <<, end with >> and contain any number of terms. One must match the value.

conjunctions These start with { , end with } and contain any number of terms. All must match the value. You are allowed to have zero terms in the conjunction, in which case it matches anything at all!

predicates There are seven prefix operators. Each should be followed by a term:

`=` For example, `= fred` must match `fred`.

`<>` For example, `<> 3` will match anything that does not match 3.

`<=>` is a 'same-type-as' test. Thus `<=> <whisky>`, where the variable `whisky` is bound to a number, will match any number.

`<, <=, >, >=` are standard numeric comparisons.

1.3.2 The actions on the right-hand side

An action also looks like a LISP list. It begins with the name of the action, this is followed the arguments of the action. The most common actions are:

make Its argument is a pattern. It creates a new working memory element. For example, `(make family ^wife juliet)`. Unspecified attributes are given a default value.

modify Usually this is followed by a number *N* referring to the working memory element which matched the *N*-th non-negated pattern on the left-hand side. Thus if this fires:

```
(p junk
    (goal setup)
    (motion ^type <x>)
    - (prevented ^type <x>)
    (route ^kind unknown)
    -->
    (modify 1 goal travel)
    (modify 3 ^kind <x>)
)
```

the `goal` and `route` working memory elements which matched on the left-hand side will be modified.

remove Usually this is followed by one or more numbers. The working memory elements which matched the relevant non-negated left-hand side patterns are deleted.

write expects a right-hand side pattern as argument, and writes it out. The special functions (`crlf`), (`tabto` *N*) and (`rjust` *N*) can be used within `write`, and have reasonably obvious meanings. Note that `rjust` (right-justify) sets up a right-justified field of the given width, for use in displaying whatever immediately follows it.

halt stops the intepreter.

bind usually expects a variable and a pattern, and binds the variable to that pattern – for instance, (bind <x> (compute <x> + 1)). If you omit the pattern, the variable is bound to a gensym'ed atom.

call calls a LISP function. The method of passing arguments depends on the version and host language of the OPS5 you are using.

There are various functions which can be used in right-hand side actions. The most useful are:

accept By default it takes input from the current input; you can give a filename as argument if you want. It reads a list (if it sees a '(' first) or a single atom. What it reads is spliced into the right-hand side action in place of the accept fuction call.

acceptline behaves like accept but reads one complete line of input, splicing it in as before. You can give a sequence of arguments; these are used if a null line is read or reading has reached the end of the input.

compute evaluates arithmetic expressions. It recognises the operators +, -, *, // (division) and \\ (modulus). Sub-expressions can be bracketed.

2 An example of OPS5

Here is a complete OPS5 program:

```
;;;
;;;                    Factorial Program in OPS-5
;;;              Simulates the stack in working memory
;;;

;;;  Stage 1. Fill the working memory with (fact 1) .. (fact n)
;;;
(p fact0 (fact {<x> = 0}) --> (remove 1) (make factorial 1 1))
;;; NB (remove 1) means delete the element that matched the first
```

```
;;;    non-negated condition. The (make factorial 1 1) means add
;;;    an element (factorial 1 1).

(p factn (fact {<n> > 0}) --> (make fact (compute <n> - 1)))
;;; NB the cumbersome {..} bit matches a symbol which is a number
;;;    larger than 0. Note also the nasty way of getting a
;;;    calculation done.

;;;  Negative number entered. Quit
(p factneg (fact {<n> < 0}) -->
    (write "Good-bye."))

;;;  Stage 2. Sweep out the unnecessary (fact k) and
;;;     (factorial k (k-1)!) and add new (factorial (k+1) k!)
;;;
(p factorial (factorial <x> <y>) (fact <x>) -->
    (remove 2)
    (remove 1)
    (make factorial (compute (<x> + 1)) (compute (<x> * <y>))))

;;;  When no more (fact k) statements are left, factorial k
;;;     has been found.
(p factorial2 (factorial <x> <y>) -(fact <x>) -->
    (write (crlf))
    (write (compute <x> - 1))
    (write "!! =")
    (write <y>)
    (write (crlf))
    (make infinifact))
;;; NB (crlf) is a built-in action component...

;;;  Called once at the beginning
;;;
(p pretty-fact (start) -->
    (write "This program accepts numbers, whose factorials")
    (write (crlf))
    (write "- lo and behold - it then computes RECURSIVELY")
    (write (crlf))
    (write (crlf))
    (remove 1)
```

```
    (make infinifact))

;;;  Start of a "loop" that reads in numbers
(p infinifact (infinifact) -->
    (remove 1)
    (write "Enter a positive number to compute its factorial,")
    (write (crlf))
    (write "or a negative one to quit")
    (write (crlf))
    (bind <x> (accept))
    (make fact <x>))

;;;  Start the whole thing rolling
(make start)
```

For more detail about OPS5 itself, and a survey of more modern production systems, see *Programming Expert Systems in OPS5* by Brownston, Farrell, Kant and Martin, pub. Addison-Wesley 1985.

Production system can be thought of as the assembly language of expert systems. They offer tremendous flexibility but no immediately useful facilities for the developer or end user. For example, you have to design and build your own explanation mechanisms. In general, production systems are most useful when the kind of knowledge you want to capture does not have a small, clean conceptual basis, and when the rule set may be large but there is no very great interdependence between the actions.