Expert Systems & Knowledge Acquisition: 1

Chris Thornton -- January 11, 1989

-- POP11 PROGRAMMING 1 (expressions and commands) ----

The course involves looking at many computational techniques. We need a general purpose programming language.

Needs to be
- high level
- easy to read
- good for expert systems
- good for learning systems

Candidates
C, Pascal - too low level
Lisp - difficult to read
Prolog - too high level (DFS is not a useful primitive in ML)

Depth first Search

Pop11 (according to Laventhol, 1987)
- structure like Pascal
- storage allocation like Lisp
- argument passing like Forth

stack
garbage collection

Pop11 is interesting in itself
- widely used for AI work in UK
- not committed to any particular AI paradigm
- implementation language for Poplog AI environment

-- Pop11 references ----------

Barrett, Ramsay & Sloman 1985;
Burton & Shadbolt 1987;
Laventhol 1987;
Pountain 1988; -- rave review in BYTE;
Ramsay & Barrett 1987;
Yazdani 1984;

-- Statements, expressions and commands ----------

A Pop11 program is a sequence of statements separated by semi-colons. Statements are either expressions or commands.

Expressions are just statements which have a value (actually, one or more values). Commands are statements which do not have a value.

Initially, we don't need to make a distinction between compiling and running a Pop11 program. We can just talk about "doing" some Pop11 code.

This can mean either (i) typing it in a top-level (Pop11 prompt is the colon) or (ii) marking and executing a range of text in an editor (e.g. Ved).

-- Numerical expressions and simple commands ---------

```
2 + 2; ==>;
```

This is a little Pop11 program made up of two statements - an expression ("2 + 2") and a command ("==>") called the "print arrow". It produces the response (i.e. output)

```
** 4
```

The print arrow acts like a semi-colon. Thus the terminating semi-colons are spurious.

```
2 + 2 ==>
** 4
```

Note that white space is ignored.

```
2
  +
2 ==>
** 4
```

Pop11 puts two asterisks before anything printed using the print arrow.

Expressions can be constructed using all the the usual mathematical operators (plus some extras).

```
X + Y      add two numbers
X - Y      subtract two numbers
- X       negate one number
X * Y      multiply two numbers
X ** Y     raise X to the power Y
X / Y      divide first number by second
X // Y     divide X by Y and return remainder and quotient.
X div Y    returns just the quotient.
X rem Y    returns the remainder on dividing X by Y
X mod Y    returns the modulus
```

Expressions can contain expressions and the usual operator precedences are implemented.

```
2 + 3 * 4 + 6 ==>
** 20
```

-- Functions ----------

Expressions can also contain function calls. A function call has the form

```
<name of function> ( <argument1>, <argument2> ...)
```

There are a number of built-in functions.

```
sqrt(25) ==>
** 5.0

min(3,2) * 6 ==>
** 12

max(3,2) + 2 ==>
** 5

abs(4.3) ==>
** 4.3

abs(-4.3) ==>
** 4.3
```

-- Variables ----------

```
vars mine;
```

Named variables are set up using the "vars" command. They can be initialised to some value by following them with an equals sign and an expression.

```
vars mine = 2 + 2;
```

The name of a variable is treated as an expression whose value is the contents of the variable.

```
mine ==>
** 4

vars yours = mine;

yours * 2 ==>
** 8
```

-- Assignments ----------

To assign a value to a variable we use the "assignment arrow" (works left to right, unlike Lisp, Pascal, C, etc.)

```
3 + 2 -> yours;
```

Pop11 first puts the value of "3 + 2" on a stack data structure (the "user stack"), then it takes it off the stack and puts it into "yours". It is possible to make Pop11 copy something off the stack without removing it. This means that one value can be assigned to two or more variables at once.

```
2 + 6 * 4 / 6 ->> yours -> mine;

yours ==>
** 6

mine ==>
** 6
```

A Pop11 program made up of four statements (two expressions, two commands):

```
10 -> mine;
mine * 2 ==>
```

The output produced is

** 20

-- Boolean expressions ----------

- Expressions can have truth values (called boolean values). These are contained in the variables called "true" and "false". Pop11 prints boolean values thus.

```
true ==>
** <true>

false ==>
** <false>
```

Pop11 has all the usual boolean operators. By using these we can construct expressions with boolean values; e.g.

```
3 = 2 ==>
** <false>

3 < 2 ==>
** <false>
```

- Other boolean operators:

```
X = Y  is true if  X has the same value as Y
X > Y              X is greater than Y
X < Y              X is less than Y
X >= Y             X is greater than or equal to Y
X <= Y             X is less than or equal to Y
X and Y            X and Y both have the value true
X or Y             either X or Y has the value true
not(X)             X is false
```

- Pop11 also provides a large number of boolean functions. For example

```
isnumber(3) ==>
** <true>

isboolean(true) ==>
** <true>

isboolean(3) ==>
** <false>
```

-- Word expressions ----------

```
mine ==>
```

Pop11 treats the "mine" bit as an expression whose value is the contents of the variable called "mine". What happens if we want to use the word itself?

- To do this we construct a special expression (a word expression) whose value is a word object. A word expression is just a sequence of ascii characters enclosed in double quotes. Thus

```
"mine" ==>
** mine
```

Word expressions can contain underscore characters.

```
"my_word" ==>
** my_word
```

Word expressions effectively construct data objects called words. Some booleans expressions involving word expressions make sense:

```
"mine" = "mine" ==>
** <true>
```

Some don't.

```
"mine" > "mine" ==>
      ^
;;; MISHAP - NUMBER(S) NEEDED
;;; INVOLVING: mine mine
;;; DOING   : > compile nextitem compile
```

(Variable names are actually word objects.)

-- List expressions ----------

List expressions effectively construct objects which are made up from sequences of values. A list expression has the form

```
[ <value1> <value2> ... ]
```

i.e. it is a sequence of arbitrary values enclosed in square brackets. The expressions normally need to be separated by spaces.

When constructing the list, Pop11 does not evaluate the individual items in the sequence; thus

```
[2 + 2] ==>
** [2 + 2]
```

We can construct lists of words as follows. Because the things which go between the square brackets are *values*, we do not need to put in word quotes.

```
[foo bang ding] ==>
** [foo bang ding]
```

-- Single hats ----------

Often we want a list to contain the value of an expression. To get this effect we need to tell Pop11 to evaluate the expression in question. We do this by enclosing the expression in brackets and preceding it with the "hat" operator ("^"). Thus

```
[foo bang ^(2 + 2)] ==>
** [foo bang 4]
```

If the expression is just the name of a variable, we don't need the round brackets.

```
[foo bang ^mine] ==>
** [foo bang 10]
```

We can get a sequence of expressions evaluated just by including more things in the round brackets. The sequence of expressions is treated just like a little Pop11 program, so we have to separate the expressions with semi-colons.

```
[foo bang ^(2 + 2; 3 + 3)] ==>
** [foo bang 4 6]
```

An alternative notation involves enclosing the Pop11 code to be evaluated in percent signs.

```
[foo bang %2 + 2; 3 + 3%] ==>
** [foo bang 4 6]
```

-- Double hats ----------

Lists can contain arbitrary values including the values of list expressions.

```
[foo bang ding] -> mine;
[1 2 ^mine 3] ==>
** [1 2 [foo bang ding] 3]
```

If we want the sequence of values in a list to be spliced into another list, we use a variant of the hat operator called the "double hat".

```
[1 2 ^^mine 3] ==>
** [1 2 foo bang ding 3]
```

If we try to use the double hat on an expression whose value is not a list, we get a mishap.

```
[1 2 ^^yours 3] ==>
        ^
;;; MISHAP - LIST NEEDED
;;; INVOLVING: 6
;;; DOING   : null dl compile nextitem compile
```

-- List functions ----------

Pop11 provides a number of functions which work with lists.

```
vars mine = [foo bang ding];

hd(mine) ==>
** foo

tl(mine) ==>
** [bang ding]

last(mine) ==>
** ding
```

```
member("bang", mine) ==>
** <true>
```

It also allows us to access the Nth element of a list. To do this we treat the variable containing the list as if it was a function which takes a numeric argument. This is called "subscripting".

```
mine(2) ==>
** bang

"hello" -> mine(2);

mine(2) ==>
** hello

mine ==>
** [foo hello ding]
```

-- Complex commands ----------

Both the assignment arrow and the print arrow are simple commands. Pop11 also provides complex commands. These always have a block structure of the form

```
<name-of-command>
  ...
  <code>
  ...
end<name-of-command>
```

-- Conditionals ----------

A fundamental complex command in Pop11 is the "if" command. This is the main conditional construct. It has the form

```
if <expression> then <code> endif;
```

or, alternatively

```
if <expression> do <code> endif;
```

In either case, the <code> will only be done if the <expression> does not have the value <false>.

```
if 2 < 3 then 10 ==> endif;
** 10

if 2 > 3 then 10 ==> endif;
```

Note that any expression whose value is not <false> is considered to be <true>. Thus

```
if 2 + 2 then 10 ==> endif;
** 10
```

"if" commands can have "elseif" parts and one "else" part. There can be any number of elseif parts.

```
vars result;

if 3 < 2 then
  3 -> result;
elseif 1 < 2 then
  1 -> result;
elseif 0 = 1 then
  0 -> result;
else
  false -> result;
endif;

result ==>
** 1
```

- Pop11 works down through the command until it finds an expression whose value is not <false>. It then does all the code which follows, up until the next "elseif", "else" or "endif".

There is also an "unless" command. This works just like an inverted "if".

```
unless <expression> do <code> endunless;
```

If <expression> has the value <false> the <code> will be executed.

-- The repeat command ----------

There are several complex commands in Pop11 which produce looping behaviour. A simple example is the "repeat" command. This has the form

```
repeat <expression> times <code> endrepeat;
```

The <code> is done a certain number of times depending on the value of <expression>; e.g.

```
repeat 5 times  2 + 2 ==>  endrepeat;
```

produces the output

```
** 4
** 4
** 4
** 4
** 4
```

<code> can be an arbitrarily long sequence of statements.

```
repeat 2 times
  2 + 2 ==>
  3 * 3 ==>
endrepeat;

** 4
** 9
** 4
** 9
```

-- The quitloop command ----------

It is possible to obtain infinite looping by just leaving out the "N times" part from the first line of the repeat command. In this case, looping can be terminated using the "quitloop" command. This command causes the looping to be stopped immediately.

```
vars n = 0;
repeat
  n + 1 -> n;
  n ==>
  if n > 5 then quitloop endif;
endrepeat;

** 1
** 2
** 3
** 4
** 5
** 6
```

The quitloop command can be used even if the "N times" part is present.

-- The while and until commands ----------

Instead of using quitloop inside repeat commands, we can use the "while" or "until" command.

```
repeat
  quitif(<expression>);
  ...
endrepeat;
```

produces the same behaviour as

```
while not(<expression>) do
  ...
endwhile;
```

and

```
until <expression> do
  ...
enduntil;
```

-- The for command ----------

- A very useful looping command is "for". This can be used in a number of different ways. The command arranges for a special variable (called the loop variable) to be set to a new value each time around the loop. The name of the loop variable is inserted immediately after the "for". Thus

```
vars n;
for n from 1 to 6 do n ==> endfor;
** 1
** 2
** 3
** 4
** 5
** 6

for n from 12 by -2 to 3 do n ==> endfor;
** 12
** 10
** 8
** 6
** 4
```

- The for command can also be used to iterate over the items in a list. In this case, each time around the loop, the next item of the list (working forwards from the front) is assigned to be the value of the loop variable.

```
vars num, list = [one two three four];
for num in list do num ==> endfor;
** one
** two
** three
** four
```

Also:

```
for num on list do num ==> endfor;
** [one two three four]
** [two three four]
** [three four]
** [four]
```

-- nextloop v. quitloop ----------

- Note that the quitloop command can be used inside repeat, while, until and for loops. It can also be used in the case where one looping command is contained inside another. To quit the Nth enclosing loop, we do quitloop(N). Thus

```
vars list number;
for list in [[1 2 3][4 5 6][7 8 9]] do
  for number in list do
    if number = 5 then quitloop(2) endif;
    number ==>
  endfor;
endfor;
** 1
** 2
** 3
** 4
```

whereas

```
for list in [[1 2 3][4 5 6][7 8 9]] do
  for number in list do
    if number = 5 then quitloop endif;
    number ==>
  endfor;
endfor;
** 1
** 2
** 3
** 4
** 7
** 8
** 9
```

A relation of quitloop is "nextloop". This works just like quitloop except that instead of bringing the looping to an end, it simply causes Pop11 to jump to the next cycle of the loop without doing anything more in the current cycle.

-- Using loops to construct lists ----------

Whenever Pop11 computes the value of an expression it is put on the stack. This means that we can construct lists by inserting expressions into loops, thus.

```
[%(for item in [7 18 49] do item * item endfor)] ==>
** [49 324 2401]

[%for item from 1 to 20 do item endfor%] ==>
** [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
```

Expert Systems & Knowledge Acquisition: 2

Chris Thornton -- January 11, 1989

-- POP11 PROGRAMMING 2 (procedures) ----------

- The "procedure" command is used to define new procedures and functions. It has the structure of a command but actually works more like an expression. That is to say, it has a value and the value is a procedure object. A procedure in Pop11 is just a sequence of statements which can be run as a unit.

```
vars next number = 0;
procedure;
  number + 1 -> number;
  number ==>
endprocedure -> next;
```

- If we put round brackets after a variable which contains a procedure, Pop11 responds by running the corresponding sequence of commands. This is referred to as "calling", "running" or "executing" the procedure.

```
next();
** 1

next();
** 2

next();
** 3
```

- If we want we can set up procedures which take inputs. To do this we simply insert a sequence of variable names in round brackets after the word "procedure" in the definition. In this case, when we call the procedure, we put a corresponding sequence of values inside the round brackets in the call. Pop11 responds by putting each value into the corresponding variable in the definition.

```
procedure(increment);
  number + increment -> number;
  number ==>
endprocedure -> next;

next(3);
** 7

next(5);
** 12

procedure(increment, n);
  number + increment -> number;
  number * n -> number;
  number ==>
endprocedure -> next;

next(2,3);
** 42
```

-- define ----------

Normally when setting up procedures in Pop11, the "define" command is used rather than "procedure".

```
define mine(x) ... enddefine;
```

is short for

```
vars mine;
procedure(x); ... endprocedure -> mine;
```

-- return ----------

- We can make a procedure behave like a built-in function by including a "return" command somewhere in the definition. This command causes Pop11 to (a) stop doing the statements making up the procedure and (b) put the value of the expression which appears in round brackets after the "return" on the stack. This allows procedure calls to be treated in the same way as calls on built-in functions.

```
define square(number);
  number * number -> number;
  return(number);
enddefine;

square(3) ==>
** 9

square(26) ==>
** 676

vars yours = square(4);

yours ==>
** 16
```

Alternatively, we can set up an explicit output parameter for a function by including another named variable after a "->", thus.

```
define square(number) -> answer;
  number * number -> answer;
enddefine;
```

For ~~variable~~ >1 returns
-> var1 -> var2 etc.

-- Recursion ----------

Procedures can make use of themselves recursively. For instance, we can define the factorial function as follows.

```
define factorial(num);
  if num = 1 then
    return(1)
  else /* recurse */
    return(num * factorial(num - 1))
  endif
enddefine;
```

Note that text appearing between the strings "/*" and "*/" is ignored by Pop11.

```
factorial(4) ==>
** 24
```

-- Local variables ----------

- Variables appearing in procedure header lines are *local* to the procedure in question. This means that they are associated with a particular call of the procedure. A local variable can have the same name as a variable which is declared outside a procedure definition. However, as far as Pop11 is concerned, the two variables are quite distinct. We can set up extra local variables by putting vars commands inside procedure definitions.

```
vars temp;
10 -> temp;

define double_then_square(number);
  vars temp;
  number * 2 -> temp;
  return(temp * temp)
enddefine;

double_then_square(3) ==>
** 36

temp ==>
** 10
```

In the case where a procedure calls itself recursively, each call has its own versions of the local variables. All versions of a given local variable are distinct.

-- Applist ----------

Sometimes programs need to process the elements of a list using a single function. For instance we might want to write a program which prints out welcoming messages for a group of people.

```
vars people = [chris fred ruth];

define say_hello(person);
  [hello ^person, nice to meet you] ==>
enddefine;

applist(people, say_hello);
** [hello chris , nice to meet you]
** [hello fred , nice to meet you]
** [hello ruth , nice to meet you]
```

-- Maplist ----------

A close relation of the applist procedure is "maplist". This takes a list and a function and returns the list which is constructed when the function is applied to all the elements of the list in turn.

```
maplist([4 16 23 89], sqrt) ==>
** [2.0 4.0 4.79583 9.43398]

maplist([4 16 23 89], double_then_square) ==>
** [64 1024 2116 31684]
```

-- Syssort ----------

- Syssort is a built-in function which works a bit like maplist. Calls on syssort have the form

```
syssort(<list>, <boolean_function>)
```

The first argument is a list, the second argument a boolean function which takes two items and returns a boolean result. If E1 and E2 are elements of the list and the value of

```
<boolean_function>(E1,E2)
```

is not <false>, then E1 will not come after E2 in the list which is returned by the function. Thus,

```
define smaller(numa, numb);
  return(numa < numb)
enddefine;

syssort([5 3 8 9 32 1 4 2], smaller) ==>
** [1 2 3 4 5 8 9 32]
```

-- Matching ----------

- One of the most powerful features of Pop11 is the pattern-matcher. This facility is packaged in the form of a boolean operator which tests whether two lists "match". The "matches" operator is usually just called the "matcher" and "matches" expressions are just called "match expressions".

If we want to test whether two lists are exactly the same we can use the "=" operator; e.g.

```
[1 2 3] = [1 2 3] ==>
** <true>

[1 2 3] = [2 2 3] ==>
** <false>
```

However, if we want to test something a little less categorical like whether or not a list ends with a certain two elements we can use the matcher thus.

```
[2 2 3] matches [= 2 3] ==>
** <true>
```

- The matcher assumes that an "=" in the second list can match an item in the first list, no matter what that item actually is. (Note that you cannot put an "=" in the first list.) There is another symbol called the "double-equals" which the matcher assumes can match a sequence of arbitrary items in the first list. Thus, if we want to check whether some list of arbitrary size ends in a certain two elements we might type

```
[foo bang ding 1 2 3] matches [== 2 3] ==>
** <true>
```

- As far as the matcher is concerned, the double-equals matches any number of items in the other list. We can put single-equals and double-equals wherever we want; e.g.

```
[foo bang ding 1 2 3] matches [== bang = 1 == 3] ==>
** <true>

[foo bang ding 1 2 3] matches [== bang = = 1 == 3] ==>
** <false>
```

- Note that by putting two separate single-equals after the "bang" in the second list we have effectively stopped it from matching any list which has "bang" and "1" but only one element in between.

Very often, we will be interested to know what things the single-equals and double-equals matched against. The matcher allows us to find this out by using two variants of the equals signs called the "single-query" (written "?") and the "double-query" (written "??").

```
[foo bang ding 1 2 3] matches [?mine bang = 1 == 3] ==>
** <true>

mine ==>
** foo
```

- As a side-effect of matching "?mine" against "foo", the matcher has put "foo" into the variable called "mine". If we use the double-query, then a list of the elements which have matched is put into the corresponding variable; e.g.

```
[foo bang ding 1 2 3] matches [??mine 1 2 3] ==>
** <true>

mine ==>
** [foo bang ding]
```

-- Restriction procedures ----------

- The single- and double-query can be used to great effect in Pop11 programs (as we will see) for "digging out" bits and pieces from list structures. However, the overall usefulness is increased still further by a feature known as "restriction procedures". The use of restriction procedures can be illustrated by considering the way in which a built-in function such as "isnumber" might be used to influence the result of a match process. The "isnumber" function takes a single expression as input and tests whether its value is a number. If it is, <true> is returned; otherwise <false> is returned. Thus

```
isnumber(2 + 2) ==>
** <true>

isnumber(3) ==>
** <true>

isnumber("foo") ==>
** <false>

isnumber([foo bang]) ==>
** <false>
```

- Let us imagine that we want to test whether some list begins with a number, ends with the word "foo", and has an arbitrary number of elements in between. We can implement this test using the matcher as follows.

```
vars list;
[1 ding bong foo] -> list;

list matches [?mine:isnumber == foo] ==>
** <true>
```

- By writing "isnumber" after the "?mine", we tell Pop11 (in effect) that the first element of the list can match against the "mine" variable provided that it is a number. Note that if it is not a number, the matcher returns <false>.

```
[ding bong foo] -> list;
list matches [?mine:isnumber == foo] ==>
** <false>
```

- Technically, putting ":" followed by the name of a function tells Pop11 that the variable can match an element provided that, when the function is called with the element provided as input, it does not return <false>. We can illustrate this by defining a new function which returns boolean values as shown in Figure 0-0. This function can then be used in match expressions as follows.

```
define is_silly_word(w);
  if w = "foo" or w = "ding" then
    return("silly")
  else
    return(false)
  endif
enddefine;

list matches [?mine:is_silly_word ==] ==>
** <true>

[sensible ding bong] -> list;

list matches [?mine:is_silly_word ==] ==>
** <false>
```

can force matcher to search list exhaustively to find match.

-- Restriction procedures can have side-effects ---------

- If the function whose name we write after the ":" returns a result which is not equal to <false> but not equal to <true> either, then the value is assigned to the variable in the place of the element which was actually provided as input. Thus

```
mine ==>
** silly
```

-- Integer restrictions ----------

- As well as function names, we can also put integers after any colon which follows a double-query or a double-equals. The idea here is that the integer forces the matcher to match the given variable against a sequence of a given length. Inserting an integer N makes sure that the variable will only match against a sequence of N elements. Thus, typing something like

```
list matches [== ??mine] ==>
** <true>
```

This has the side effect of putting a list containing all the elements of the matched list into the variable "mine". Whereas

```
list matches [== ??mine:2] ==>
```


= equivalent to Unix?
== " " Unix ?

** <true>

returns <true> and has the side effect of putting a list containing just the final 2 elements into the variable mine.

```
mine ==>
** [ding bong]
```

- Lists which contain symbols which mean certain things to the matcher are usually called patterns. Note that, for convenience, a match expression whose arguments are not list expressions works just like an "=" expression. Thus,

```
2 matches 3 ==>
** <false>

"foo" matches "foo" ==>
** <true>
```

runs the matcher and throws away the result

-- The matcher arrow ----------

- As we have seen, the Pop11 matcher can have useful side-effects. In some cases we may know that a list will definitely match a pattern but want to obtain the side-effects of the match anyway. In this case we can use a command "-->" which is called the "matcher arrow". An expression featuring the matcher arrow between two list expressions works just like a match expression except that it does not have a value. Thus

```
[foo 2 3 ] --> [?mine:is_silly_word ==];
```

gets the word "silly" assigned to the variable called "mine".

```
mine ==>
** silly
```

Alternatively, we could use the matcher arrow to assign values to the variables called "mine" and "yours" simultaneously.

```
[1 2] --> [?mine ?yours];

mine ==>
** 1

yours ==>
** 2
```

-- The database ----------

- As has already been shown, Pop11 is a good language for dealing with list structures. Because, programs will often need to set up and then process many different lists, the language provides a general place in which lists can be stored and accessed. This is called the "database". In fact the database is itself a list but normally, the lists that are of interest are the ones which are in the database list. Pop11 provides simple commands for adding lists into the database, for taking them out and for finding all examples of lists which match a certain pattern. To create a new database from scratch we can use an ordinary list expression and an assignment; e.g.

```
[[1 2 3] [foo bang] [noddy bigears]] -> database;
```

The normal way of adding lists into the database list, involves using a Pop11 command called "add". An "add" command consists of the word "add" followed by a list expression appearing between round brackets.

```
[] -> database;
add([Chris Thornton Brighton Sussex]);
```

- This causes the list in brackets to be added to the database list (which is initialised to be an empty list). Since the database list is stored in a variable called "database", we can print out the contents of the database using the print arrow in the usual way.

```
database ==>
** [[Chris Thornton Brighton Sussex]]
```

We can add more lists by executing a sequence of "add" commands; e.g.

```
add([Fred Bloggs Lewes Sussex]);
add([Simple Simon Dover Kent]);
add([Margaret Thatcher London]);
add([John Smith Brighton Sussex]);
```

-- Present ----------

- We can use the built-in "present" function to test whether there is a list in the database which matches a certain pattern. The value of a call on "present" is <true> if there is a list in the database matching the pattern given as first input, and <false> otherwise. Thus

```
present([== Brighton Sussex]) ==>
** <true>

present([== Surrey]) ==>
** <false>
```

If we want to check that the database contains a set of patterns then we can use the function "allpresent", thus.

```
allpresent([[Chris Thornton ==][Fred Bloggs ==]]) ==>
** <true>
```

-- Foreach ----------

- Another very useful command is "foreach". This is like "for" except that instead of iterating over every element of a list, it iterates over every element of the database which matches a certain pattern. Every time it finds a match, it puts the list in question into a special variable called "it". Thus

```
foreach [== Brighton Sussex] do it ==> endforeach;
** [John Smith Brighton Sussex]
** [Chris Thornton Brighton Sussex]

foreach [== Sussex] do it ==> endforeach;
** [John Smith Brighton Sussex]
** [Fred Bloggs Lewes Sussex]
** [Chris Thornton Brighton Sussex]
```

If we want the "foreach" command to iterate over the sublists of some other list we can use a command of the form

```
foreach <pattern> in <other list> do <commands> endforeach;
```

- Note that there is version of "present" called "isin" which works for ordinary lists. This is actually an operator like "matches"; expressions involving the "isin" operator have the value <true> or <false> depending whether the list (i.e. the value of the list expression) which appears on the right can be matched with one of the sublists of the list which appears on the left. Thus,

```
[= 2 =] isin [[4 5 6][7 8 9][1 2 3]] ==>
** <true>
```

-- Tracing ----------

- There is a command which tells Pop11 to print out information each time a procedure is called. In the case where a procedure calls itself recursively, this printing can be very helpful.

To ask for information to be printed out every time "factorial" is called do

```
trace factorial;
```

- This is referred to as "tracing" the function.

```
factorial(4) ==>

>factorial 4
!>factorial 3
!!>factorial 2
!!!>factorial 1
!!!<factorial 1
!!<factorial 2
!<factorial 6
<factorial 24
** 24
```

- Once we have traced a function Pop11 responds to calls on that function by printing out certain information. When we type

```
factorial(4)
```

Pop11 prints out

```
>factorial 4
```

- The ">" indicates that Pop11 is starting to evaluate a call, the word is just the name of the called function, and any items which appear after are just the inputs which have been provided in the call. If in the process of evaluating the call, a new call on a function is made, then Pop11 acts virtually the same way except it precedes all the printing with an exclamation mark. This indicates that the new call is *nested* inside the original

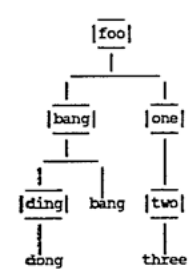
call. For each level of nesting we get one more exclamation mark.

- If ever we want to stop Pop11 printing our information about function calls we can use the "untrace" command. This is handled in exactly the same way as the "trace" command: we type the word "untrace" followed by one or more function names. Pop11 then turns tracing off for these functions; i.e., it stops producing trace output every time they are called.

-- Showtree ----------

- The trace feature allows us to display the structure of a sequence of nested function calls. Pop11 also provides a useful feature which allows us to display and inspect the structure of a list. As was noted above, lists can contain lists as elements. These nested lists can contain other lists, which can contain still more lists. If we have such a structure we can see what it looks like by using a Pop11 command called "showtree". This command simply creates a graphic description in which the nesting structure is shown as a tree structure. For reasons that will become obvious, the tree is always drawn upside-down with its root sticking up in the air. Note that, in most implementations, to make the "showtree" command available we need to type a special "lib" command as follows.

```
lib showtree;
[foo
   [bang [ding dong] bang]
   [one [two three]]] -> list;
showtree(list);
```



A possible interaction using the "showtree" command is shown above.

-- Readline ----------

- Another useful function which is built-in to Pop11 is "readline". This takes no inputs at all. When it is called it prints out a single question mark and then makes Pop11 wait until you type something. It lets you type in whatever you want but when you press the <RETURN> key the function collects up all the words which you have typed and returns them in a list as the result of the original call on "readline". Thus we can type

```
readline() -> mine;
? here are a few random words

mine ==>
** [here are a few random words]
```

- Quite often we will want to use "readline" in order to find out whether the user agrees to something or not. In this case we can use the built-in function "yesno". This takes a list as input. It prints out the list, and then calls "readline" to read the user's response. If the user responds either with "yes" or "no", the "yesno" function returns <true> or <false> respectively. If the user types anything other than "yes" or "no" the function prints out a complaint and goes through the whole process again. It will keep doing this until the user types either a "yes" or a "no". Thus

```
yesno([do you agree that the world is flat?]) -> mine;

** [do you agree that the world is flat ?]
? what a stupid question
** [please answer either yes or no]
** [do you agree that the world is flat ?]
? I would prefer not to
** [please answer either yes or no]
** [do you agree that the world is flat ?]
? ok, yes
** [please answer either yes or no]
** [do you agree that the world is flat ?]
? yes

mine ==>
** <true>
```

== Tutorial exercise (for tutorial in week 2) -------

To do these exercises you will find it useful to read up to the end of chapter 3 of Barrett, Ramsay & Sloman (BRS) OR read to the end of chapter 3 of TEACH PRIMER (POPLOG teach file) or to the end of chapter 3 of Laventhol.

1. Using examples explain the difference in meaning between the following seven symbols
(i) -> (ii) => (iii) ->> (iv) ==> (v) - (vi) = (vii) >

2. Explain the difference between the following five objects
(i) "a" (ii) 'a' (iii) 'a (iv) a (v) [a]

3. What is the difference between in meaning of the following three expressions
(i) 1 + 2 * 3 (ii) (1 + 2) * 3 (iii) 1 + (2 * 3)

4. Explain the difference between the following eight (including MISHAPS if any)
(i) sqrt(25) (ii) [sqrt(25)] (iii) 'sqrt(25)' (iv) "sqrt(25)"
(v) sqrt("25") (vi) sqrt([25]) (vii) sqrt((25)) (viii) sqrt

5. What do each of the following twelve mean? If any are incorrect, explain why. (If you know what MISHAP message is created in a particular case, explain that too.)

```
(i)   44 -> fred;
(ii)  4 4 -> fred;
(iii) 44 - > fred;
(iv)  sqrt(49) -> maryjane;
(v)   sqrt ( 49 ) ->   maryjane;
(vi)  sqrt(
      49
      )->
      maryjane
      ;
(vii) sqrt(49)->mary jane;
(viii) sqrt(49) -> mary_jane;
(ix)  sqrt(49) -> mary-jane;
(x)   44 -> "fred";
(xi)  fred -> 36;
(xii) 7 -> sqrt(49);
```

== Other exercises 1: Simple procedures ----------

For these exercises you will need a telephone list in the following general form, possibly with further entries:

[[mary jones] 479625 [fred smith] 249371 [mike fret] 567923] -> telephones;

1. Write four different versions of a procedure named print_phonelist that takes a telephone list as argument and prints out on a separate line each person's name and their telephone number. The procedure should not return a result. The four versions should make use of:
(i) repeat ... times ... endrepeat (ii) for ... in ... do ... endfor
(iii) until ... do ... enduntil (iv) while ... do ... endwhile

2. Write two different versions of a procedure get_name that given a telephone number and telephone list as arguments returns the corresponding name as its result e.g.

```
get_name(249371,telephones) =>
** [fred smith]
```

One version should use (i) the matcher, the other (ii) any looping construct.

3. Write a procedure named get_first_name that expects the same arguments as get_name (in Question 2) but simply returns the person's first name. The new procedure should call get_name as a sub-procedure.

4. To demonstrate the idea of "garbage in garbage out" write a procedure named average that takes a single argument, a telephone list, and returns the arithmetic average (to the nearest whole number) of the telephone numbers it finds!

5. Write a procedure named find_average_person that takes a telephone list as argument and returns either the name of the first person who happens to have the same telephone number as the average, or if such a person cannot be found returns the list [no luck] as its result.

6. Write a procedure named get_nums of no arguments that makes use of readline to interactively build up a telephone list in the form above and when complete return it as its result. When the user types "no more" it should stop e.g.

```
get_nums() -> newlist;
** [name]
? mary haddock
** [number]
? 12345
** [name]
? jack sprat
** [number]
? 22222
** [name]
? no more
newlist =>
** [[jack sprat] 22222 [mary haddock] 12345]
```

== Other exercises 2: Using the database ----------

For the following exercises you will need store telephone entries in the POP-11 database. Each entry should be of the following general form

[[<name of person>][<male/female><telephone number><school>][<interests>]]

e.g.

[[mary jones][female 477238 MAPS][welding yoga dance archery]]
or
[[fred alfred smith][male 345262 COGS][yoga cooking programming]]
or
[[jack higgins][male no_phone EAPS][dance knitting cars]]
or
[[susan augustina louise bloggs][female 12345 COGS][football]]

1. Create a POP-11 database with at least 10 such telephone entries, either by assigning a list containing sub-lists as above to the variable database, or by using the procedure add or the procedure alladd.

2. Write a procedure show_details that takes two arguments namely gender and school. The procedure should print out all the entries in the database, one per line, which have both the given gender and the given school. e.g.

```
show_details("male","COGS");
** [[fred alfred smith][male 345262 COGS][yoga cooking programming]]
```

3. Write a procedure add_clients that prompts the user for details of a new client, checks to see if the client's name is not already in the database, prompts for the gender, telephone number, school and interests and then adds the entry to the database. The procedure should loop until the user types nomore in response to the request for a new client name. e.g.

```
add_clients();
** [what is the name of the next client]
? mary jones
** [we already have details of this person]
** [what is the name of the next client]
? mark brown
** [what is the gender of mark brown]
? male
** [what is the telephone number of mark brown]
? 34567
** [what is the school of mark brown]
? AFRAS
** [what are the interests of mark brown]
hang_gliding raffia reading
** [what is the name of the next client]
? nomore
```

Check that your procedure adds new entries to the database in the correct form.

4. Write a procedure called find_friend that takes a list of interests as its only argument and prints out, one per line, every entry in the database that has at least one interest in common with the list supplied e.g.

```
find_friend([food yoga knitting]);
** [mary jones 477238]
** [fred alfred smith 345262]
** [jack higgins no_phone]
```

5. Write a procedure called pairs that prints out the names of every compatible pair of persons in the database. To be compatible each person of a pair must have at least one interest in common with the other person. A person cannot form a pair with him/herself. e.g.

```
pairs();
** [mary jones AND fred alfred smith]
** [mary jones AND jack higgins]
** [fred alfred smith AND mary jones]
** [jack higgins AND mary jones]
```

== Other exercises 3: Writing recursive procedures ---

1. (i) Write a recursive procedure called count_down that given an integer as argument counts down from that integer to 0 e.g.

```
count_down(4);
** 4
** 3
** 2
** 1
** 0
```

(ii) By adjusting count_down make a new procedure count_down_up which first counts down to 0 and up from 0 to the given number.

2. Write a recursive procedure named intersect that takes two lists as arguments and returns another list which contains only those elements which occur in both the input lists. If there are no such elements the procedure should return []. e.g.

```
intersect([tom dick harry],[mary susan jane]) =>
** []
intersect([a b c d e f],[e f g a]) =>
** [a e f]
```

3. Write a recursive procedure named check that takes an arbitrarily deeply nested list as its argument. The procedure should return true if there is any integer anywhere in any of the lists or sub-lists and otherwise return false. e.g.

```
check([a [b c [[e f]]g] h i]) =>
** <false>
check([a [b c [[e f]]5] h i]) =>
** <true>
```

4. Write 3 versions of a procedure called cubeall which takes a list of integers as its argument and returns a list containing the cubes of all the numbers e.g.

```
cubeall([4 2 6 1]) =>
** [64 8 216 1]
```

(i) using maplist (ii) using recursion (iii) using iteration

5. Adjust your answer to question 8 to produce a recursive procedure named censor. It should take the same kind of argument as check but rather than return simply true or false, it should return the nested list that it was given with any integers replaced by the word "censored". e.g.

```
check([a [b c [[e f]]g] h i]) =>
** [a [b c [[e f]] g] h i])
check([a [b c [[99 f]]5] h i]) =>
** [a [b c [[censored f]] censored] h i]
```