AN INTRODUCTION TO POP-11 PROGRAMMING FOR AI

S.Lynch,

© Copyright S.Lynch 1997. All rights reserved.
This document may be copied & distributed free of charge for educational purposes only.

Section 1

AN INTRODUCTION TO POP-11

These notes provide an introduction to POP-11, they are pitched at someone who has already had an introduction to programming and understands the use of things like functions & variable scope. In particular, I have assumed that the reader has some interest in programming & the mechanics of programming languages.

The first section gives a general introduction to POP as a list processing language. The next section introduces some more advanced features & uses them to develop some functions to deal with simple symbolic representations often used in AI applications. The last section introduces the use of lexical closures & demonstrates how languages like POP can use an object oriented approach without needing object oriented language facilities.

POP-11 was originally designed to be used as a teaching language for novice programmers. It has evolved into a powerful, general purpose language favoured by many for constructing AI programs. It can be used as a symbolic, list-processing language but also includes all the facilities, typically offered by 3GLs for number processing. As well as lists & symbols POP provides a range of data types for strings, arrays, and records as well as some less common data types like fractions & complex numbers. POP has a powerful pattern matcher and operators to deal with sets & association lists.

POP programs can be developed using different approaches, I have taken a functional approach through most of these notes. Occasionally you will find a reference to XSL. XSL is a utility package which provides some extra POP facilities, ensure you have a copy of XSL.

POP appears as an interpreted language so, as well as typing complete programs using an editor, it is possible to interact directly with the POP system. In this case POP statements are evaluated as soon as they are entered. More accurately, POP is not interpreted but compiled by a single step compiler, to a machine-independent Pcode which is immediately evaluated. In POP-speak this is known as *immediate mode*.

The integrated environment containing the POP-11 compiler is known as Poplog. It also contains an editor and a help system as well as compilers/interpreters for Common Lisp, Prolog & ML.

Much of what follows is *immediate mode* dialog with the poplog system, the **:** character at the start of each line is the prompt given by poplog, the **=>** sign (called a print arrow) gets POP to print out the result of last thing that was done. When a line starts with ****** it has been printed by a print arrow. Comments start with **;;;** and end at the end of a line.

THE BASICS

POP deals mostly with two types of data - symbols & lists. Symbols are things like numbers and words.

```
:
: 12 =>          ;;; this is a symbol with a numeric value
** 12
:
: "banana" =>    ;;; this is a symbol whose value is a word.
** banana
:
```

Note the use of the quotes around banana, without them POP would assume the word banana referred to something like a variable.

Lists are collections of symbols between square brackets, you don't need quotes round symbols when they are in lists.

```
:
: [ the cat sat on the mat ] =>    ;;; this is a list of 6 words
** [the cat sat on the mat]
:
```

this is a list whose 2nd item is a list

```
:
: [ the [cat sat] on the mat ] =>
** [the [cat sat] on the mat]
:
```

any list can hold different types of items

```
:
: [ Liverpool 1, Middlesbrough 5 ] =>
** [Liverpool 1 , Middlesbrough 5]
:
```

the empty list is a special list, it is sometimes called nil

```
:
: [] =>
** []
:
: nil =>
** []
:
: nil = [] =>        ;;; this is a test for equality
** <true>
:
: nil = "banana" =>     ;;; this is another one
** <false>
:
```

SOME POP-11 FN.S FOR MESSING ABOUT WITH LISTS

A fn is called with its name followed by its arguments between round brackets.

hd (head) - this is a Fn. which returns the first item in a list

```
:
: hd( [a b c d] ) =>
** a
:
: hd( [[a b] [c d]] ) =>
** [a b]
:
: hd( hd( [[a b][c d]] ) ) =>
** a
:
```

tl (tail) - this returns the list with its head removed

```
:
: tl( [a b c d] ) =>
** [b c d]
:
: tl( [a] ) =>
** []
:
: tl( tl( [a b c d] ) ) =>
** [c d]
:
```

head & tail are two fns which pull lists apart, concatenate & cons are infix operators which squash lists together.

<> (concatenate) - merges 2 lists

```
:
: [a b c] <> [d e f] =>
** [a b c d e f]
:
: [a b c] <> nil =>
** [a b c]
:
: nil <> nil <> nil =>
** []
:
```

:: (cons) - adds an item onto the front of a list

```
:
: "a" :: [b c d] =>
** [a b c d]
:
: [a b c] :: [d e f] =>
** [[a b c] d e f]
:
```

brackets can be used to order infix operations

```
: "a" :: ( "b" :: ( "c" :: nil ) ) =>
** [a b c]
:
: nil :: nil :: nil =>      ;;; which cons is carried out first ?
** [[[]]]
:
```

null is a boolean fn which tests a list to see if it is equal to nil

```
:
: null( [a b c] ) =>
** <false>
:
: null( [] ) =>
** <true>
:
: null( nil ) =>
** <true>
:
```

VARIABLES & ASSIGNMENT

In general, if you use an undeclared variable then pop declares it for you and tells you that it has been declared.

```
:
: [do be do be do] -> alist;
;;; DECLARING VARIABLE alist
:
```

The above expression assigns the list [do be do be do] to a new variable called 'alist'. Assignment is done using an assignment arrow -> and is specified from left to right. This is different to assignment expressions in nearly all other languages which are the wrong way round. The semi-colon is used to mark the end of the assignment expression. I havn't used semi-colons before because all expressions have been followed by a print arrow.

```
:
: alist =>
** [do be do be do]
:
: hd( alist ) =>
** do
:
:
```

More examples, but this time I will declare some variables first. Variables can be declared in a couple of different ways but for now I'll use the word **vars**.

```
:
: vars num1, num2;
: 4 -> num1;
: 7 -> num2;
: num1 + num2 -> num3;
;;; DECLARING VARIABLE num3
:
: num3 =>
** 11
:
```

POP has all the normal arithmetic operators **+ - / *** and all of the normal logical operators **= < > >= <=** the only unusual one is not equals which is done with the symbol **/=**

```
:
: num3 < 20 =>
** <true>
:
: num3 /= 11 =>
** <false>
:
```

THE STACK

Underlying POP-11 are two stacks; the procedure stack, which is mainly hidden from you - the programmer, and the user stack. Passing parameters, assignment & nearly all other operations make use of the user stack. Understanding how the user stack works will help you to develop POP code;

Consider a typical assignment statement

```
5 -> x;
```

What actually goes on during the processing of this statement can be broken into two parts:

```
:
: 5;        ;;; put value 5 on Top Of user Stack (TOS)
: -> x;     ;;; remove top item from user stack & assign it to x;
:
```

Expressions that produce values have there results put on the user stack. A simple example of such an expression is ' 5; ' in the above example.

```
:
: 5 + 3;            ;;; put value 8 on the stack
: "fred";           ;;; "fred" goes on the stack
: hd([a b c d]);    ;;; "a" on stack
:
: ==>               ;;; this is the pretty print arrow, it only
                    ;;; removes & prints the top item on the
                    ;;; stack, not all the stacked items
** a
:
: ==>
** fred
:
: ==>
** 8
:
```

Another example with assignment:

```
:
: 4;         ;;; stack value 4
: 3;         ;;; stack value 3
: -> x;      ;;; unstack top item (3) & assign to x
: -> y;      ;;; unstack top item (4) & assign to y
:
: x =>
** 3
: y =>
** 4
:
: ;;; to swap x & y
: x; y -> x -> y;
:
: x =>
** 4
: y =>
** 3
:
```

DEFINING FUNCTIONS

There are two different ways to write a function definition depending on the method you want to use for passing the results of the function back to its *caller* (the program or function which calls it). One method directly uses the stack to return values, the other uses specified variables. The choice between the two styles is partly personal taste but often the stack-based method is simpler. For the first few examples I will use both approaches, after that I'll use only the one that seems most natural.

Functions are defined using the POP word **define**. Using the stack for return values **define** has the following form:

```
define <fn-name> ( ...list of arguments... );
       <fn-body>
       enddefine;
```

This example is of a function that takes a number as its only argument & returns a result of its argument plus 1.

```
:
: define add1( number );
:   number + 1;
:   enddefine;
:
: add1( 12 ) =>
** 13
:
: vars n1, n2;
: 3 -> n1;
: add1( n1 ) =>
** 4
:
: n1 =>
** 3
:
: add1( 5 ) -> n2;
: n2 =>
** 6
```

Look back at the function declaration. Notice how the result of the expression `number + 1` is not assigned to anything. The result is just left on the user stack. Now think about the call to add1 above `add1( 5 ) -> n2;` As outlined before, the assignment mechanism just lifts the last value off the stack & binds it to the named variable so `add1( 5 )` evaluates leaving the number 6 on the stack, `-> n2` lifts the 6 off the stack, binding it to n2.

Why consider the mechanism for value passing/assignment in this detail? The answer is that (apart from being interesting) good use of the stack can make some bits of code easier to write or execute more efficiently, you can't make effective use of it if you don't clearly understand what it can do for you.

Defining functions method 2: explicitly using variables to return results. In this case **define** is used like this:

```
define <fn-name> ( ...list of args... ) -> result-var ;
       <fn-body>
       enddefine;
```

In this case the value returned from the function is not the value left on the stack but the value of its *result-var*.

```
:
: define add1( number ) -> new_valu;
:   number + 1 -> new_valu;
:   enddefine;
:
: add1( 4 ) =>
** 5
:
: add1( -2 ) =>
** -1
:
: vars result;
: add1( add1( 7 ) ) -> result;
: result =>
** 9
:
```

With add1 defined in this way the POP system takes the value in the variable `new_valu` & dumps it on the stack when the function ends. People who mostly use Pascal tend to prefer this style of declaration when they first start using POP.

Note: POP does not clean up the stack when a function finishes. It is the programmer's reponsibility to ensure that rogue values are not left lying around on the stack.

CONDITIONAL FORMS

POP uses an 'if-then-else-endif' structure as its main conditional form. Its layout is quite flexible but is typically something like:

```
if     <condition-1> then
       <actions-1>
elseif <condition-2> then
       <actions-2>
else
       <actions-3>
endif;
```

The 'elseif' and 'else' blocks are both optional, you can use as many 'elseif' blocks as you like but can obviously only have one 'else' block (note that 'elseif' is one word not two).

Here is a sample fn using an if statement, it takes two lists as arguments & returns the longer of those lists as its result. Length is a POP fn which takes a list as its only arg & returns the length of that list. Informally the logic is: if list1 is longer than list2 then the answer is list1 otherwise the answer is list2.

Method 1.

```
:
: define longer( list1, list2 );
:   if length( list1 ) > length( list2 ) then
:       list1;
:   else
:       list2;
:   endif;
:   enddefine;
:
: longer( [l a g e r], [g u i n e s s] ) =>
** [g u i n e s s]
:
: longer( [k i p p e r], [c o d] ) =>
** [k i p p e r]
:
```

Method 2.

```
:
: define longer( list1, list2 ) -> longest;
:   ;;; result returned through variable called longest
:   if length( list1 ) > length( list2 ) then
:       list1 -> longest;
:   else
:       list2 -> longest;
:   endif;
:   enddefine;
:
: longer( [a b c], [p q r s] ) =>
** [p q r s]
:
```

REPETITION IN POP

Consider writing a fn sum_upto which takes one positive integer arg N and returns the sum of all integers from 1 to N. One way to build this fn would be to use a loop & accumulate the sum as the loop was executed. Another way to build the fn is based on the premis that:

sum from 1 to N = N + sum from 1 to (N-1)
AND sum when N = 0 is 0

This can be constructed in POP as follows:

```
:
: define sum_upto( N );
:   if N = 0 then
:       0
:   else
:       N + sum_upto( N-1 )
:   endif;
:   enddefine;
:
: sum_upto( 12 ) =>
** 78
:
```

A note about syntax: the semi-colon after statements immediately before 'else' & 'endif' are optional, there doesn't seem to be any accepted standard so please yourself.

It is easyer to see how sum_upto works by following its progress using the trace command. Each time the fn is called trace prints a > sign, the fn name and its args. When the fn terminates trace prints a <, the fn name & its result. For fns which call themselves, ! characters are also printed to show how 'deep' a particular fn is in the calling sequence.

```
:
: trace sum_upto;
:
: sum_upto( 4 ) =>
> sum_upto 4
!> sum_upto 3
!!> sum_upto 2
!!!> sum_upto 1
!!!!> sum_upto 0
!!!!< sum_upto 0
!!!< sum_upto 1
!!< sum_upto 3
!< sum_upto 6
< sum_upto 10
** 10
:
```

Defining sum_upto using the 2nd method:

```
:
: define sum_upto( N ) -> result;
:   ;;; calculates the sum of integers from 1 to N
:   if N = 0 then
:       0 -> result;
:   else
:       N + sum_upto( N-1 ) -> result;
:   endif;
:   enddefine;
:
: sum_upto( 5 ) =>
** 15
:
```

Note that the information given when tracing a fn does not depend on how it was defined.

```
:
: trace sum_upto;
:
: sum_upto( 4 ) =>
> sum_upto 4
!> sum_upto 3
!!> sum_upto 2
!!!> sum_upto 1
!!!!> sum_upto 0
!!!!< sum_upto 0
!!!< sum_upto 1
!!< sum_upto 3
!< sum_upto 6
< sum_upto 10
** 10
:
:
```

From now onwards I will only write one version of a function, either using a return variable or using the stack - whichever seems most natural for the example I'm using. This next example is a function which takes a list & returns it's length (like the predefined POP function called length).

The function length can be designed in a similar way to sum_upto, based on the premis that:

the length of a list is 1 + the length of its tail AND the length of an empty list is 0

in POP

```
:
: define list_len( lis );
:   if null( lis ) then
:       0;
:   else
:       1 + list_len( tl(lis) );
:   endif;
:   enddefine;
:
: list_len( [blah blah blah] ) =>
** 3
:
: trace list_len;
:
: list_len( [haddock herring hake halibut] ) =>
> list_len [haddock herring hake halibut]
!> list_len [herring hake halibut]
!!> list_len [hake halibut]
!!!> list_len [halibut]
!!!!> list_len []
!!!!< list_len 0
!!!< list_len 1
!!< list_len 2
!< list_len 3
< list_len 4
** 4
:
```

This style of repetitive strategy (called recursion) is a convenient way to build functions which perform some action on each item in a list without disrupting the order of the list.

Consider a function to increment every number in a list of numbers, so

inc_list([1 2 7 5]) => [2 3 8 6]

A verbose algorithm

```
inc_list( lis )
    vars newhead, newtail, newlist      ;;; do without these later
    if null( lis ) then                 ;;; if empty list
        return: nil                     ;;;    return empty list
    else                                ;;; if non empty list
        hd(lis) + 1 -> newhead          ;;;    add 1 to head
        inc_list( tl(lis) ) -> newtail  ;;;    inc_list deals with tail
        newhead :: newtail -> newlist   ;;;    cons newhead & newtail
        return: newlist                 ;;;    return result
    endif
```

Consise algorithm

```
inc_list( lis )
    if null( lis ) then
        return: nil
    else
        return: ( hd(lis) + 1 ) :: inc_list( tl(lis) )
    endif
```

in POP:

```
:
: define inc_list( lis ) -> newlist;
:   ;;; increments each number in a list of numbers
:   if null( lis ) then
:       nil -> newlist;
:   else
:       ( hd(lis) + 1 ) :: inc_list( tl(lis) ) -> newlist;
:   endif;
:   enddefine;
:
: inc_list( [1 2 7 5] ) =>
** [2 3 8 6]
:
: trace inc_list;
:
: inc_list( [1 2 7 5] ) =>
> inc_list [1 2 7 5]
!> inc_list [2 7 5]
!!> inc_list [7 5]
!!!> inc_list [5]
!!!!> inc_list []
!!!!< inc_list []
!!!< inc_list [6]
!!< inc_list [8 6]
!< inc_list [3 8 6]
< inc_list [2 3 8 6]
** [2 3 8 6]
:
```

A MORE INTERESTING EXAMPLE: A WORLD OF BLOCKS

This example works on a commonly used structure for expressing relationships between named objects. The structure 'blocks' (below) describes the way objects are laid out on a table top. Each entry in 'blocks' is a triple of the form [relation object-name value], like [color b1 red] for example. The objects are called b1, b2 ... b6.

The problem this example tackles is that of building some general purpose fns for retrieving information from such a structure. Giving suitable responses to requests for things like "the names of all cubes ".

```
vars blocks;
    [   [isa    b1  cube]       [isa    b2  wedge]
        [isa    b3  cube]       [isa    b4  wedge]
        [isa    b5  cube]       [isa    b6  wedge]
        [color  b1  red ]       [color  b2  red  ]
        [color  b3  red ]       [color  b4  blue ]
        [color  b5  blue]       [color  b6  blue ]
        [on     b1  table]      [on     b2  table]
        [on     b5  table]
    ] -> blocks;
```

The first step is to build a couple of useful fns. The first three functions pick out different parts of a triple. It would be easy to get by without these functions but it's good practice to include them & I will make use of some of them later in this booklet.

```
define relnof( triple );
    triple(1);
    enddefine;

define nameof( triple );
    triple(2);
    enddefine;

define valueof( triple );
    triple(3);
    enddefine;

:
: relnof( [isa ball sphere] ) =>
** isa
:
: nameof( [isa ball sphere] ) =>
** ball
:
: valueof( [isa ball sphere] ) =>
** sphere
:
```

The next couple of functions check the value of different parts of a triple:

```
define reln_eq( triple, reln );
    relnof(triple) = reln;
    enddefine;

define value_eq( triple, val );
    valueof(triple) = val;
    enddefine;

:
: reln_eq( [isa ball sphere], "isa" ) =>
** <true>
:
: reln_eq( [isa ball sphere], "color" ) =>
** <false>
:
: value_eq( [isa ball sphere], "sphere" ) =>
** <true>
:
: value_eq( [isa ball sphere], "wedge" ) =>
** <false>
:
```

Now for the main function. A recursive function called select which takes a list of triples, a relation & a value & returns a list of object names. The names returned are for objects which have triples of the given type. So:

```
select( blocks, "color", "blue" )
```

will return the names of all blue objects.

```
define select( triples, reln, val );
    if null( triples ) then
        nil;
    elseif reln_eq( hd(triples), reln )
            and value_eq( hd(triples), val ) then
        nameof(hd(triples)) :: select( tl(triples), reln, val );
    else
        select( tl(triples), reln, val );
    endif;
    enddefine;

: select( blocks, "color", "red" ) =>
** [b1 b2 b3]
:
: select( blocks, "on", "table" ) =>
** [b1 b2 b5]
:
```

ITERATION THROUGH LISTS

Recursion can often be the best approach for repetition with lists but POP also supports various iterative forms.

The contruction:

```
for <var> in <list> do
    <actions>
endfor;
```

sets up an iteration where <var> takes on successive items in <list> and <actions> are evaluated each time the loop cycles.

EG 1:

```
: vars item;
: for item in [a [b c] d] do
:   item =>     ;;; print item
: endfor;
** a
** [b c]
** d
:
```

EG 2: a function to sum up a list of numbers. This is an example where it is more natural to return the function value through a variable rather than using the stack. The function uses a local variable, notice how it is declared with a vars statement after the fn header.

```
define sumlist( nums ) -> sum;
    ;;; sum up No.s in nums, result returned through sum
    vars item;                  ;;; local variable
    0 -> sum;                   ;;; initialise sum to 0
    for item in nums do
        item + sum -> sum;      ;;; add item to sum
    endfor;
    enddefine;

:
: sumlist( [1 3 5 9] ) =>
** 18
:
```

Other iterative forms available include for-endfor loops which increment a numeric variable between two bounds, until-do-enduntil, while and repeat forms. See online documentation for details.

MAPPING FUNCTIONS

One particularly useful function in POP organises a style of iteration for you. The function maplist takes two args: a list and a function to be applied to each member of that list, maplist returns a new list containing the all the results of its function arg.

```
:
: define add1( number );
:   number + 1;
:   enddefine;
:
: maplist( [1 3 7 11], add1 ) =>
** [2 4 8 12]
:
: ;;; isnumber returns true if its one arg is numeric
:
: maplist( [a 1 b 2 c d 3], isnumber ) =>
** [<false> <true> <false> <true> <false> <false> <true>]
:
```

Two other functions which apply functions to lists are REMOVE_IF and REMOVE_IF_NOT defined in XSL (note their names are in upper-case). Both functions take a list & a function as arguments.

REMOVE_IF applys its function argument to each item in its list argument & removes those items for which the function returns non-false.

```
:
: REMOVE_IF( [2 cod and 3 chips 1 with vinegar], isnumber ) =>
** [cod and chips with vinegar]
:
```

REMOVE_IF_NOT is similar to REMOVE_IF but it removes items for which its function returns false. If REMOVE_IF_NOT's function does not return false then REMOVE_IF_NOT does one of two things. If its function returns true for an item in its list then REMOVE_IF_NOT preserves that item. If its function returns something other than true REMOVE_IF_NOT replaces that item with the value returned.This makes REMOVE_IF_NOT suprisingly versatile & it will be used often in examples covered later in these notes.

```
:
: REMOVE_IF_NOT( [2 cod and 3 chips 1 with vinegar], isnumber )
=>
** [2 3 1]
:
: define silly( X );
:   if islist( X ) then
:       rev( X )
:   else
:       false
:   endif;
:   enddefine;
:
: REMOVE_IF_NOT([hello [John Pete Sue] how was [Jan Feb March]],
:      silly)=>
** [[Sue Pete John] [March Feb Jan]]
:
```

ANONYMOUS FUNCTIONS

When we use define to to declare a function the function is explicitly given a name. It is also possible to declare functions without giving them a name. There are various reasons why it might be convenient to do this. One example is for use as an argument to maplist or REMOVE_IF etc.

As an example I'm going to try removing the negative numbers from a list of numbers. One way to do this is by defining a boolean function called 'negative' then using it in REMOVE_IF.

```
define negative( X );
    X < 0;
    enddefine;

:
: [34 -6 7 -21 -9 14] -> numberlist;
: REMOVE_IF( numberlist, negative ) =>
** [34 7 14]
:
```

Another way of doing this uses an anonymous function within the REMOVE_IF call. Anonymous functions are built in POP using the 'procedure' syntax.

```
:
: REMOVE_IF( numberlist, procedure( X );
:                            X < 0;
:                         endprocedure
:          ) =>
** [34 7 14]
:
```

Note the syntax: the words **procedure** & **endprocedure** instead of **define enddefine** and the function name which normally follows the word **define** is ommited. The arguments list & body are the same as with a **define** form:

```
procedure( argument-list );
    statements
endprocedure;
```

As with functions built using define you can specify result variables.

Another example.

```
:
: maplist( [1 3 7 14], procedure(n) -> result;
:                          n+1 -> result;
:                       endprocedure
:        ) =>
** [2 4 8 15]
:
```

THE BLOCKS EXAMPLE

Back to the world of blocks example introduced a few pages ago. In this example a list of triples describing objects on a table is held in a structure called **blocks**. If you can't remember this example or the use of functions like **reln_eq** then turn back & re-read that section before going on.

This time select is defined using REMOVE_IF_NOT and returns whole triples rather than just object names. As you will see, using **maplist** with **nameof** gives the object names for these triples.

```
define select( triples, reln, val );
    REMOVE_IF_NOT( triples,
                   procedure(t);
                      reln_eq( t, reln ) and value_eq( t, val );
                   endprocedure );
    enddefine;

:
: select( blocks, "color", "red" ) =>
** [[color b1 red] [color b2 red] [color b3 red]]
:
: maplist( select( blocks, "color", "red" ), nameof ) =>
** [b1 b2 b3]
:
```

MISHAP MESSAGES

As you may have discovered by now errors in POP are called 'mishaps', this is presumably because the designers thought the term 'mishap' might not depress us as much as the term 'error'. All error messages are difficult to read at first (even ones that are called something else) but compared with a lot of languages POP's messages are not too bad. Look at this example:

```
:
: add1( "haddock" ) =>

;;; MISHAP - NUMBER(S) NEEDED
;;; INVOLVING:  haddock 1
;;; DOING    :  + add1 compile Im runproc popvedcommand compile
```

- The 1st line, starting 'mishap', indicates the type of mishap or the reason why the mishap was generated.
- The second line (involving) lists the operands/values/etc that were being used when the mishap occurred.
- The last line (doing) lists the functions/controls active at the time the mishap occurred. As a general rule, reading left to right on this list you can ignore everything after the first compile. In the above example that leaves `+ add1` which means when the mishap occurred POP was trying to carry out a + operation inside a function called add1.

PUSHING VALUES INTO LISTS

Usually anything between square brackets is treated as if it had quote characters around it. The symbols ^ and ^^ push values into lists or, put another way, cause POP to evaluate some given symbol. ^ followed by a variable name (or an expression between round brackets) pushes out the value of the variable (or expression).

```
:
: vars x;
: "cat" -> x;
: [the ^x sat] =>
** [the cat sat]
:
: [cat chased] -> x;
: [the ^x the rat] =>
** [the [cat chased] the rat]
:
```

Note that ^x treats the value of x as an item. ^^ is similar except that it
treats the following value as a list of items:

```
:
: [cat chased] -> x;
: [the ^^x the rat] =>
** [the cat chased the rat]
:
```

There is another way to evaluate items within a list structure, that is by enclosing the part to be evaluated by % signs. This is often more readable when it comes to evaluating expressions.

```
:
: [start % hd( [a b c d] ) % end] =>
** [start a end]
:
```

Multiple values can be put between % signs separated by commas

```
:
: [% hd( [a b c] ), tl( [x y z] ) %] =>
** [a [y z]]
:
```

This approach can be to build lists in a variety of ways. The following example uses an iterative form to construct a list of numbers from 1 to 10. If you are unsure how this example works read the section on the user stack.

```
:
: [%  for i from 1 to 10 do
:        i;
:     endfor;
: %] =>
** [1 2 3 4 5 6 7 8 9 10]
:
```

THE BLOCKS EXAMPLE AGAIN

Back to the world of blocks example introduced a few pages ago & revisited in the *Anonymous Functions* section. In this example a list of triples describing objects on a table is held in a structure called **blocks**. If you can't remember this example then turn back & re-read the section which introduced it.

This time select is defined using the technique described above of enclosing a for loop in list & evaluating it.

```
define select( triples, reln, val );
    vars t;
    [%  for t in triples do
            if reln_eq( t, reln ) and value_eq( t, val ) then
                nameof( t );
            endif;
        endfor;
    %];
    enddefine;

:
: ;;; iterative without matches
:
: select( blocks, "color", "red" ) =>
** [b1 b2 b3]
:
: select( blocks, "isa", "wedge" ) =>
** [b2 b4 b6]
:
```

THE MATCHER

POP-11 provides a powerfull pattern matcher, the following notes detail some of its features. Pattern matching provides a mechanism for pulling items out of list structures. In its most basic form the matcher is like the equals comparison between lists.

```
:
: [a b c] = [a b c] =>
** <true>
: [x y z] = [a b c] =>
** <false>
:
: [a b c] matches [a b c] =>
** <true>
: [x y z] matches [a b c] =>
** <false>
:
```

Various special symbols can be used on the right hand side (RHS) of the matches expression. The RHS is known as the pattern. These notes go through some of these symbols.

The item wild card symbol: =

This symbol (the = sign) will match with any item in a corresponding position in the list on the left hand side (LHS) of the matches expression. The = wild card matches with one and only one item. The pattern [a = c] will match with any three item list starting with "a" & ending with "c".

```
:
: [a b c] matches [a = c] =>
** <true>
:
: [a [p q r] c] matches [a = c] =>
** <true>
:
: [a p q r c] matches [a = c] =>
** <false>
:
```

The list wild card symbol: ==

This wild card will match with zero or more items in the LHS of the matches expression. The pattern [a == c] will match with any list starting with "a" & ending with "c".

```
:
: [a b c] matches [a == c] =>
** <true>
:
: [a c] matches [a == c] =>
** <true>
:
: [a p q r c] matches [a == c] =>
** <true>
:
```

The item match-in symbol: ?

When the ? symbol appears in the pattern the matcher takes the symbol/word after it to be a variable name. The ?*variablename* part of the pattern works a bit like the = wild card except that the item matched against on the LHS is assigned to the named variable. The pattern [a ?X c] will match with anythin that [a = c] will match with but the variable X will take on the value of the 2nd item in the matched list.

```
:
: vars X;
: [a b c] matches [a ?X c] =>
** <true>
: X =>
** b
:
: [a [p q r] c] matches [a ?X c] =>
** <true>
: X =>
** [p q r]
:
: [a p q r c] matches [a ?X c] =>
** <false>
:
```

There are a couple of things you need to note about about matching variables:

1. You cannot use lexically scoped variables (see later) since they will not be in scope of the matcher.
2. Do not assume anything about a matching variable when matching fails (ie: when matches returns false).

The list match-in symbol: ??

This is similar to the item match-in symbol except that the variable is assigned zero or more items & gets a list structure as a value.

```
:
: [a b c] matches [a ??X c] =>
** <true>
: X =>
** [b]
:
: [a c] matches [a ??X c] =>
** <true>
: X =>
** []
:
: [a p q r c] matches [a ??X c] =>
** <true>
: X =>
** [p q r]
:
```

Note: the ^ and ^^ symbols may be used in matching patterns in both LHS & RHS.

One more thing, if you know the shape of the structure you wish to match against and you are confident that matches will always return true. If you are interested in doing the pattern matching purely for its side effects then you can use the structure assignment arrow **-->** This is an infix operator which functions like matches except that it does not return true if the match succeeds (and it gives you a mishap message if it fails).

THE BLOCKS EXAMPLE USING MATCHES

Another go at the blocks example. This time select will be written using the matcher. I'll show two versions this time. One using mapping functions & the other using an iterative form.

```
define select( triples, pattern );
     maplist( REMOVE_IF_NOT( triples,
                            procedure(t);
                                t matches pattern;
                            endprocedure ),
              nameof );
     enddefine;

define select( triples, pattern );
    vars t;
    [%  for t in triples do
            if t matches pattern then
                nameof( t );
            endif;
        endfor;
    %];
    enddefine;

:
: select( blocks, [color = red] ) =>
** [b1 b2 b3]
:
```

The next function, **moveable**, makes use of select. It returns true or false depending on whether an object (whose state is described by a set of triples) can be moved. An object can be moved if (i) it exists and (ii) it has no object on top of it.

```
define moveable( triples, object );
    not( null( select( triples, [isa ^object =] )))
      and null( select( triples, [on = ^object] ));
    enddefine;

:
: moveable( blocks, "b6" ) =>
** <true>
:
: moveable( blocks, "table" ) =>
** <false>
:
```

SOME MORE USEFUL POP FNS

member - takes two args, an item and a list. The result is true if the item arg occurs somewhere in the list, otherwise the result is false.

```
:
: member( "spam", [egg beans spam and chips] ) =>
** <true>
:
: member( "sausage", [egg beans spam and chips] ) =>
** <false>
:
```

pr - print, takes one arg (which can be just about anything) and prints it, without the ** you get with => and without throwing a newline. If you want to throw a newline then 'pr(newline);' will do it.

```
:
: pr(1) ;
1: pr("banana");    ;;; '1' was printed by pr, ':' is the prompt
banana: pr( [the cat sat] ); pr(newline);
[the cat sat]
:
```

readline - this fn takes no args, it reads a line of text from the keyboard into a list & retruns this list as its reult. The ? at the start of the line is the prompt from readline.

```
:
: readline() -> input;
? mines a pint
:
: input =>
** [mines a pint]
:
```

Another function, **readitem**, reads a single item from the keyboard.

dest - Until loops often work with lists by doing something to their head & then cycling round with their tail. The POP fn "dest" does a hd & a tl operation simultaneously, producing two results. So the two lines:

```
:
: hd( list1 ) -> itm;
: tl( list1 ) -> list2;
:
```

Are equivalent to one call to dest:

```
:
: dest( list1 ) -> list2 -> itm;
:
```

Note the order of the assignments after the dest call, the first value picked up is the tail, the second one is the head.

NUMERICAL INDEXING OF LISTS.

A brief note about numeric indexing of lists: POP allows lists to be accessed by an index, like arrays are in other langauges. This approach should only be used when it would be messy to use other approaches, it is an inefficient way of accessing lists & rarely helps the readability of code. (If you want to use arrays, they are available in POP, see on line documentation).

```
:
: [a b c](1) =>
** a
:
: vars list;
: [a [b c] d] -> list;
: list(2) =>
** [b c]
:
: list(2)(1) =>
** b
:
```

ANOTHER EXAMPLE USING THE MATCHER.

This example is based round the use of the pattern matcher. It is a very superficial attempt at an Elisa type program. Elisa was one of a few programs written in the late 60's which attempted to do something with English sentences. Elisa behaved like a psychotherapist who asked a load of questions but gave little advice. Some people apparently believed that Elisa understood what they typed in. In fact the program just used a set of patterns to come up with suitable replies to the sentences they entered (to be fair, Elisa used a few natty programming tricks and while everybody scoffs about it now, it generated a lot of discussion).

The basis for this example is a set of pattern pairs. If a sentence input at the keyboard matches the 1st part of a pattern then the 2nd part is given as the program's response. A typical pattern pair might be something like:

```
[ [ I like ??a ]   [ what do like about ^^a ]  ]
```

This would match with the sentence: `[ I like cold pizza ]`
and produce the response: `[ what do like about cold pizza ]`

There is a bit of a problem with specifying matching pairs exactly like this. As soon as POP sees the ^ type symbols it evaluates the variables that follow them. But the variables will not have appropriate values until the first part of the pattern pair has been matched with.

```
:
: [ [My name is ?a] [Hello ^a] ] -> mlist;
:
: mlist =>
** [[My name is ? a] [Hello <undef a>]]
:
: [My name is Sid] matches mlist(1) =>
** <true>
:
: a =>
** Sid
:
: mlist(2) =>
** [Hello <undef a>]
:
```

I hope you appreciate the problem from the above example. What we need to do is delay the evaluation of the second part of the pattern pair until the match has occured. Evaluation can be prevented by making this second part a string. Strings are not covered in any detail in these notes but a string is anything between single quote characters. Strings can be evaluated using eval

```
:
: [ [My name is ?a] '[Hello ^a]' ] -> mlist;
:
: mlist =>
** [[My name is ? a] [Hello ^a]]
:
: datakey( mlist(2) ) =>     ;;; datakey returns an object's type
** <key string>
:
: a =>
** Sid
:
: eval( mlist(2) ) =>
** [Hello Sid]
:
```

Getting back to the original problem, the first step is to define a few pattern pairs (Note that the last one will match with anything) and a function to generate a response given a sentence & a list of pattern pairs.

```
vars a, b, matchlist;

[
   [   [ I like ??a ]   '[ what do like about ^^a ]'          ]
   [   [ ??a are ??b ]  '[ you think ^^a are ^^b ? ]'         ]
   [   [ I am ??a ]     '[ why do you think you are ^^a ? ]' ]
   [   [ == ]           '[ how very interesting ]'            ]
]  -> matchlist;

define get_reply( sentence, matchlist );
   ;;; this finds a match for sentence and
   ;;; returns the corresponding reply
   until sentence matches hd( matchlist )(1) do
       tl( matchlist ) -> matchlist;
   enduntil;
   eval( hd( matchlist )(2) );
   enddefine;

:
: get_reply( [I like cold pizza], matchlist ) =>
** [what do like about cold pizza]
:
: get_reply( [No specific match for this one], matchlist ) =>
** [how very interesting]
:
```

Finally a procedure to sit between the user & the matching routine. This one terminates when it is given a blank line of input.

```
define talk();
   ;;; this sits in between get_reply and
   ;;; whoever is using it
   vars input_line;
   readline() -> input_line;
   until null( input_line ) do
       get_reply( input_line, matchlist ) =>
       readline() -> input_line;
   enduntil;
   enddefine;

:
: talk();
? I am a fried egg
** [why do you think you are a fried egg ?]
? I like fried eggs
** [what do like about fried eggs]
? fried eggs are really great
** [you think fried eggs are really great ?]
? sometimes, but sometimes I am not so sure
** [how very interesting]
?
:
```

To set the record straight I should emphasise that this approach is **not** the basis for modern natural language processing systems.

USING RULES - ANOTHER MATCHER EXAMPLE

This example introduces a few more features of the matcher. The matcher has a lot of features not covered in this booklet - check out the on-line help for more details.

The next functions work to apply simple rules. A rule is a structure which could state (for example) that "if an animal has hair then it is a mammal" or, as in the following example,

if X eats meat then X is a carnivore

Rules have two parts: an antecedent & a consequent. The antecedent states what must be true for a rule to proceed (eg: X eats meat) and the consequent states what the rule can infer (eg: X is a carnivore).

Rules work on a set of known facts and tend to add more facts - that is often their purpose. A set of facts could be something like animals below:

```
vars animals =
   [[eats hyena  meat ]
    [eats rabbit grass]
    [eats sheep  grass]
    [eats tiger  meat ]];
```

The first stage in applying a rule is to try to match its antecedent against the set of facts. This can be done with a matches expression using wild-cards or using **isin** a matcher operator which checks to see if a pattern is present anywhere in a list of lists.

```
: animals matches [== [eats sheep grass] ==] =>
** <true>
:
: animals matches [== [eats ?x grass] ==] =>
** <true>
:
: x =>
** rabbit
:
: [eats ?x meat] isin animals =>
** <true>
:
: x =>
** hyena
:
```

Below are two versions of **applyrule**, a function which takes a set of facts, an antecedent & a consequent. **applyrule** returns the set of facts with the consequent added to them if its antecedent matched. Notice that the consequents are held as strings then evaluated. It is necessary to do this for the same reasons given in the Elisa example.

The problem with the first version of **applyrule** is that it can only find (at most) one match for its antecedent. The second version corrects this by using **foreach** - another matcher facility. If you have understood things so far you should not have too much trouble working out the meaning of **foreach**. If you have problems look in the online help.

```
define applyrule1( facts, antecedent, consequent ) -> facts;
    if antecedent isin facts then
        eval(consequent) :: facts -> facts;
    endif;
    enddefine;

:
: applyrule1( animals, [eats ?x meat], '[isa ^x carnivor]' ) ==>
** [[isa hyena carnivor]
    [eats hyena meat]
    [eats rabbit grass]
    [eats sheep grass]
    [eats tiger meat]]
:

define applyrule2( facts, antecedent, consequent ) -> facts;
    foreach antecedent in facts do
        eval(consequent) :: facts -> facts;
    endforeach;
    enddefine;

:
: applyrule2( animals, [eats ?x meat], '[isa ^x carnivor]' ) ==>
** [[isa tiger carnivor]
    [isa hyena carnivor]
    [eats hyena meat]
    [eats rabbit grass]
    [eats sheep grass]
    [eats tiger meat]]
:
```

Section 2

SETS

It is often useful to deal with sets (orderless collections of elements). Some languages provide a specific set data-type. In list based languages it is generally more convenient to have the facility to treat lists as sets when we want to. This allows us to use all the standard list operations on a set structure. XSL provides infix operators which carry out set operations using lists. These operators are briefly described below.

It is important to realise that while any set can be used as a list, not all lists can be used like sets. Two things about sets are crucial:

1. Sets are *orderless*. XSL's set operators do not guarantee to preserve the order of the elements in a set.
2. Any element should only appear once in each set. The set operators do not guarantee consistent operation if a set has duplicate entries. MAKESET is a function which removes duplicate entries from a list & thereby forms it into a set.

Each of the set operators described below is an infix operator made up of two or more symbols, starting with a # sign.

Operator	Example
SET EQUALS #= compares two sets for equality	: : [a b c d] #= [d a c b] => ** <true> : : [a b c d e] #= [d a c b] => ** <false> :
SET NOT EQUALS #/= compares two sets for non-equality	: : [a b c d] #/= [d a c b] =>** <false> : : [a b c d e] #/= [d a c b] => ** <true> :
STRICT SUPERSET #> s1 #> s2 is true if s1 is a non-equal superset of s2	: : [a b c d e] #> [d a c b] => ** <true> : : [a b c d] #> [d a c b] => ** <false> :
SUPERSET #>= s1 #> s2 is true if s1 is a superset of (maybe equal to) s2. superset is a faster operation than strict superset.	: : [a b c d e] #>= [d a c b] => ** <true> : : [a b c d] #>= [d a c b] => ** <true> :
STRICT SUBSET #< s1 #< s2 is true if s1 is a non-equal subset of s2. s1 #< s2 is equivalent to s2 #> s1.	See example for #>

SUBSET #<= s1 #<= s2 is true if s1 is a subset of (maybe equal to) s2.s1 #<= s2 is equivalent to s2 #>= s1.subset is a faster operation than strict subset.	See example for #>=
SET DIFFERENCE #- s1 #- s2 returns the set of elements in s1 which are not in s2.	: : [a b c d e] #- [g f e d c] => ** [a b] : : [g f e d c] #- [a b c d e] => ** [g f] :
SET UNION #+ The union of two sets. S1 #+ S2 is the set containing all elements that are in S1 and/or S2. If performance of set operations is important then it is a little more efficient if the smaller set is the first operand.	: : [a b c d e] #+ [g f e d c] => ** [a b g f e d c] :
SET INTERSECTION #* The intersection of two sets. S1 #* S2 is the set containing all elements that are in both S1 and S2.	: : [a b c d e] #* [g f e d c] => ** [c d e] : : [a b c d] #* [e f g] => ** [] :
INCLUDES ELEMENT #>> S #>> E if E is an element of S.	: : [a b c d] #>> "c" => ** <true> : : [a b c d] #>> "h" => ** <false> :
IS ELEMENT OF #<< E #<< S if E is an element of S. E #<< S is equivalent to S #>> E.	See example for #>>
SET ELEMENT REMOVE #-- S #-- E is the set S with the element E removed. If E is not a member of S then S #--E = S	: : [a b c d] #-- "b" => ** [a c d] :
SET ELEMENT ADD #++ S #++ E is the set S with the element E added. If E is a member of S then S #++ E = S	: : [a b c] #++ "x" => ** [x a b c] :
MAKESET Removes duplicate entries from a list, making it more suitable for set operations. This routine is slow with large sets.	: : MAKESET([a b c c d d a b b g]) : => ** [g d c b a] :

AN EXAMPLE USING SETS

This example deals with accessing a structure known as an association list to find the names of those objects which have certain specified attributes. Association lists are covered in more detail later in this booklet. For this example it is enough to think of an association list as a series of entries, each having a name and a list of attributes. A typical association list might be something like 'birds' below.

```
vars birds =
    [   [budgie      [color  yellow]
                     [eats   seed  ]
                     [size   small ]
        ]
        [sparrow     [size   small]
                     [color  brown]
                     [eats   worms]
        ]
        [robin       [size   small ]
                     [eats   worms ]
                     [color  redish]
        ]
    ];
```

The problem is to develop a fn entry_names which takes an association list and a list of attributes (an attribute could be [color yellow] for example) & checks to see which objects in the association list possess all of those attributes. The functions built below treat the list of attributes as a set. If the attributes of an object in the association list include all the attributes given to the function as its 2nd argument then the name of that object will be returned. Two versions of the same function are given. One uses an iterative approach uses & the other uses a mapping function.

```
define entry_names( entries, attribs );
    /* ITERATIVE */
    vars e;
    [%  for e in entries do
            if tl(e) #>= attribs then
                hd(e);
            endif;
        endfor;
    %];
    enddefine;

define entry_names( entries, attribs );
    /* MAPPING */
    REMOVE_IF_NOT( entries,
                   procedure(x);
                      tl(x) #>= attribs and hd(x);
                   endprocedure );
    enddefine;
:
: entry_names( birds, [[size small] [eats worms]] ) =>
** [sparrow robin]
:
```

THE BLOCKS EXAMPLE USING SETS

If you cannot remember the blocks example refer to earlier sections of this booklet.

This time we will extend the capabilities of the blocks functions by looking for more than one type of triple, giving suitable responses to requests for things like "the names of all red cubes on the table".

Blocks has been set up as follows:

```
vars blocks;
    [   [isa    b1  cube]       [isa    b2  wedge]
        [isa    b3  cube]       [isa    b4  wedge]
        [isa    b5  cube]       [isa    b6  wedge]
        [color  b1  red ]       [color  b2  red  ]
        [color  b3  red ]       [color  b4  blue ]
        [color  b5  blue]       [color  b6  blue ]
        [on     b1  table]      [on     b2  table]
        [on     b5  table]
    ] -> blocks;
```

The function 'Select' has been built (as it was in the section dealing with matches) to take a list of entries (like 'blocks') and a matching pattern (like [color = red]) and returns the object names for all the entries which match the pattern.

```
define select( triples, pattern );
    vars t;
    [%  for t in triples do
            if t matches pattern then
                nameof( t );
            endif;
        endfor;
    %];
    enddefine;

:
: select( blocks, [color = red] ) =>
** [b1 b2 b3]
:
: ;;; get the names of blue objects which are wedges
: select( blocks, [color = blue] )
:   #* select( blocks, [isa = wedge] ) =>
** [b4 b6]
:
```

If our final goal is to produce a fn which will return the names of objects fullfilling a number of different criteria then we need to repeatedly carry out set intersection operations, starting initially with the set of all objects. Simply using maplist with blocks & the objnam fn will give duplications of block names, MAKESET removes these duplicate entries.

```
:
: MAKESET( maplist( blocks, objnam ) ) =>
** [b6 b5 b4 b3 b2 b1]
:
```

In the function **which_objects** below, the criteria argument is a list of patterns which will match against triples.

```
define which_objects( entrys, criteria ) -> objset;
    vars pattern;
    ;;; objset initially contains all object names
    MAKESET( maplist( blocks, objnam ) ) -> objset;
    for pattern in criteria do
        maplist( select( blocks, pattern ), objnam ) #* objset
            -> objset;
    endfor;
    enddefine;

:
: which_objects( blocks, [ [color = blue] [isa = wedge] ] ) =>
** [b4 b6]
:
: which_objects( blocks, [ [color = red] [on = table] ] ) =>
** [b1 b2]
:
: which_objects( blocks,
:                [[color = red] [on = table] [isa = cube] ]) =>
** [b1]
:
```

AND & OR

The use of **and** & **or** is similar to their use in most other languages but is not restricted to dealing with only true & false values. For the purpose of conditional expressions POP considers everything other than false to be a value of true (ie: not false). This leads to definitions of **and** & **or** that may be subtly different to what you expect. Before looking at the examples which follow look at the definitions in the boxes below.

X and Y	**X or Y**

evaluate X
IF X = false THEN
   return false
ELSE
   evaluate Y
   return Y
	evaluate X
	IF X ≠ false THEN
	   return X
	ELSE
	   evaluate Y
	   return Y

Notice that the second operand will only be evaluated if it is necessary to do so.

```
:
: false or "banana" =>
** banana
:
: "cod" and "chips" and "peas" =>
** peas
:
: "mango" or "banana" =>
** mango
:
```

Working like this **and** & **or** are surprisingly useful operators. The next example redefines the member function.

member2(item, list)

> item is a member of list if:
> list is not empty AND either
> 1: item is equal to the hd of list OR
> 2: item is a member of the tl of list

Using this logic member can be defined as follows. Note that rather than working out the logic required to find out if item *is* a member of list we have instead specified what it means for item to *be* a member of list.

```
define member2( item, list );
    list /= nil
    and ( item = hd(list)
          or member2( item, tl(list) )  )
    enddefine;

:
: member2( "roti", [idli dosai roti vadai] ) =>
** <true>
:
: member2( "toast", [idli dosai roti vadai] ) =>
** <false>
:
```

ASSOCIATION LISTS

Arrays are a numerically indexed data structure. Lists can be numerically indexed but for many purposes it is more useful to index lists symbolically. An association list is a symbolically indexed list. It is structured as a list of associations. Each association is a list whose head is the association name & whose tail is its value.

```
: vars bus;
: [ [color red] [capacity 50] [make leyland] ] -> bus;
```

bus is now an association list of three associations. Individual values can be accessed using the infix operator ##

```
: bus ## "color" =>
** [red]
:
: bus ## "make" =>
** [leyland]
:


vars alist;
[   [car    [color      rust]
            [capacity   5   ]
            [make       VW  ]    ]
    [van    [color      blue]
            [capacity   7   ]
            [make       ford]    ]
    [bike   [capacity   2   ]
            [color      black]
            [make       triumph]    ]
] -> alist;

:
: alist ## "bike" =>
** [[capacity 2] [color black] [make triumph]]
:
: alist ## "bike" ## "color" =>
** [black]
:
```

Time to introduce a bit of terminology: associations, like [color rust], are often known as *slots*. The things that slots are associated with, like car or bike, are often known as *objects*. One other thing to note: if you use the **##** operator to access a slot which does not exist it returns a value of false.

To take things a step further, consider the next association list which holds some details about rabbits & foxes and carnivors & herbivors. It also includes Peter who is a rabbit & Freddy who is a fox. The implication is that fox is the name of a class of objects (as is carnivor, rabbit etc) and Freddy is an *instance* of that class.

```
vars animals;
[   [mammal      [has_parts  bones hair  ]   ]
    [herbivor    [a_kind_of  mammal      ]
                 [eats       plants      ]   ]
    [rabbit      [habitat    burrow      ]
                 [eats       grass       ]   ]
    [peter       [a_kind_of  rabbit      ]   ]
    [carnivor    [has_parts  sharp_teeth ]
                 [eats       rabbit      ]   ]
    [fox         [a_kind_of  carnivor    ]
                 [habitat    lair        ]   ]
    [freddy      [a_kind_of  fox         ]   ]
] -> animals;
```

In this type of structure, it is usual for most of the details about any particular instance to be found attached to the association of its class (or the class of its class, or the class of the class of its class and so on). As an example think about a likely value for the ‘eats’ slot for Freddy. The most likely value is [eats rabbit] because carnivor eats rabbit and fox is a_kind_of carnivor and Freddy is a_kind_of fox.

The mechanism that retrieves information about an *instance* from its *class* is called inheritance, you have probably heard about the idea before.

Keeping things simple, if you are looking for the value of a particular slot (eg: ‘eats’) for an object like Freddy, the first step is to see if there is an eats slot within Freddy’s own association list. In the example above there isn’t, so

```
:
: animals ## "freddy" ## "eats" =>
** <false>
:
```

If no such slot is present (implied by the false value) then the trick is to follow the a_kind_of link from Freddy to Freddy’s class name (fox) & continue looking for the required slot from there. The inheritance mechanism stops following a_kind_of links when either it finds the value it’s looking for or it runs out of a_kind_of links. If we were looking for an ‘annual-salary’ slot for Freddy we’d run out of links before we found a slot value.

Before developing an inheritance function there are two things to add. First there is a convenient operator that returns a single valued association as a word rather than a list containing a word. This is the ### operator:

```
:
: animals ## "freddy" ## "a_kind_of" =>
** [fox]
:
: animals ## "freddy" ### "a_kind_of" =>
** fox
:
```

Second, the value returned for a non-existent a_kind_of link (not surprisingly) is false.

```
:
: animals ## "mammal" ### "a_kind_of" =>
** <false>
:
```

This is important as it will provide the basis for determining if a_kind_of links have been exhausted.

What follows is an algorithm for an inherit function which takes 3 arguments:

net — an association list (often thought of as a network)
object — an object name (eg: Freddy)
slot — a slot name (eg: eats)

```
inherit( net, object, slot )
        if a_kind_of links are exhausted then
                false
        else
                if net ## object ## slot /= false then
                        net ## object ## slot
                else
                        inherit( net, net ## object ### "a_kind_of", slot )
```

This algorithm can be simplified (or shortened depending on your point of view). This simplification is based around three points:

1. When *a_kind_of* links are exhausted object = false.

2. The construction `if A /= false then A else B`
   is equivalent to `A or B`

3. The construction `if object = false then false else blah-blah-blah`
   is equivalent to `object and blah-blah-blah`

Given these points, inherit can be written:

```
define inherit( net, object, slot );
    object and
        (net ## object ## slot
        or inherit(net, net ## object ### "a_kind_of", slot) );
    enddefine;

:
: inherit( animals, "peter", "eats" ) =>
** [grass]
:
: inherit( animals, "freddy", "has_parts" ) =>
** [sharp_teeth]
:: inherit( animals, "freddy", "eats" ) =>
** [rabbit]
:
```

The recursive nature of inherit is illustrated by tracing its activity. To make the output of trace more readable I have edited it so ***animals*** replaces the full association list that is printed by trace.

```
:
: trace inherit;
:
: inherit( animals, "freddy", "eats" ) =>
> inherit animals freddy eats
!> inherit animals fox eats
!!> inherit animals carnivor eats
!!< inherit [rabbit]
!< inherit [rabbit]
< inherit [rabbit]
** [rabbit]
:
```

An inheritance mechanism & an association list network can be made much more sophisticated. If you are interested in finding out more about this kind of thing then try getting hold of a book on Artificial Intelligence & looking up Knowledge representation. You'll find that people often use ***isa*** links for connecting instances to their classes and ***ako*** (short for a_kind_of) for linking classes to super-classes. You'll also find that the assumption that inheritance always takes place through classes & super-classes is not correct. Assuming Gary is an instance of the class Person it would not make sense to get a value of a Salary slot for Gary by inheriting through the Person class. It would be better to look at the Salary slots of Gary's colleages. That is just one example of many - read an Artificial Intelligence book.

VARIABLES & TYPES.

You may have noticed that POP variables are not declared to be of any particular type. This is not because they are untyped but because they are not type restricted. In fact variables can change type and POP will co-erce variables (or 'cast' them in C terminology) into the type it considers most appropriate. A variable's type is checked when it used. Try to add a number to word & you'll get an error.

Check through the next few lines which experiment with variable types.

```
:
: vars X = "hello";
: [this is a list] -> X;
:
: 5 + 4 -> X;
: X =>
** 9
:
: hd -> X;
: X =>
** <procedure hd>
:
: X( [garlic naan and raita] ) =>
** garlic
:
: define apply_a_fn( fn, arg );
:   fn( arg );
:   enddefine;
:
: apply_a_fn( hd, [Jack & Jill & Sally] ) =>
** Jack
:
```

These next examples demonstrate number type coercion. In particular you should note the use of the fraction type.

```
:
: 4 -> X;
: X =>
** 4
:
: X * 3.14 -> X;
: X =>
** 12.56
:
: 4 -> X;
: X / 5 -> X;
: X * 10 -> X;
: X =>
** 8
:
: 4 -> X;
: X / 5 -> X;
: X =>
** 4_/5
:
```

POP also has a complex number type - if you want to know more about complex numbers ask a mathematician. NB: sqrt is POP's square root function.

```
:
: sqrt( -4 ) =>
** 0.0_+:2.0
:
: ;;; POP finds coercion of complex numbers difficult
: sqrt( -4 ) -> X;
: X * X =>
** -4.0_+:0.0
:
```

MULTIPLE RETURN VALUES

You'll notice that the dest function returns 2 values. If you've only used C or Pascal finding a language that allows functions to return multiple values will be new to you. It's easy enough to define your own POP functions to return more than one result either by leaving more values on the stack when the function terminates or by listing more return variables in the define form.

```
:
: define stupid() -> a -> b;
:     "first" -> a;
:     "second" -> b;
:     enddefine;
:
: vars r1, r2;
:
: stupid() -> r1 -> r2;
: r1 =>
** first
: r2 =>
** second
:
```

A word of caution:

1. it is rarely necessary to return more than 2 (or 3) values.
2. if all of your functions end up with 2 or 3 return values you should check your design (you may have been a bit careless).
3. more return values often mean more chance of bugs in your code.

As a more useful example I'll define the dest function. Remember what it does:

```
dest( list ) -> tl-of-list -> hd-of-list
```

eg: `dest([a b c d]) -> [b c d] -> "a"`

Here is the code to define this function:

```
    define dest2( lis ) -> tail -> head;
        vars head, tail;
        hd( lis ) -> head;
        tl( lis ) -> tail;
        enddefine;

:
: vars h, t;
: dest2( [spam egg chips beans] ) -> t -> h;
: h =>
** spam
: t =>
** [egg chips beans]
:
```

MORE ABOUT THE STACK

The stack can be used in a variety of ways. One of its uses is to take the place of a local variable (this can increase the efficiency of bits of code but tends to compromise readablity). The next example finds the sum of a few numbers without using a variable to hold the sum. The basis for this example is understanding the way the stack is used with expressions involving infix operators. The compiler translates an expression like `A + B` into a sequence of Pcode statements roughly like this:

```
stack the value of A
stack the value of B
apply + operator
```

The + operator takes the last two values off the stack, adds them & puts the result back on the stack. We can rewrite the expression like `A + B` so it makes some of the stack operations more obvious:

```
A;
+ B;
```

We can also have any number of expressions between the A; and the + as long as the value of A is on the top of the stack when + is used the result will be the same. IE:

```
A;
[this looks odd] -> alist;
+ B;
```

You will probably need to work through the next piece of code carefully to understand exactly how it works.

```
:
: 0;    ;;; stack 0
: for i from 1 to 5 do
:     + i;      ;;; add i to TOS value & stack result
: endfor;
: =>            ;;; print value on stack
** 15
:
```

Non-destructive assignment using **->>**

```
:
: vars a, b, c;
: 5;            ;;; stack 5
: 3 ->> a;      ;;; stack 3, copy TOS to a
: -> b;         ;;; unstack & assign TOS to b
: -> c;         ;;; unstack & assign TOS to c
:
: a =>
** 3
: b =>
** 3
: c =>
** 5
:
```

Non-destructive assignments are useful in a variety of ways, the following example shows their use in a conditional statement.

```
define sum_nums( lis ) -> sum;
    ;;; this fn calculates the sum of any numbers in a list
    vars n;
    0 -> sum;
    until null( lis ) do
        ;;; break lis into hd & tl
        ;;; leaving hd on stack for isnumber
        if isnumber( dest(lis) -> lis ->> n ) then
            n + sum -> sum;
        endif;
    enduntil;
    enddefine;

:
: sum_nums( [John ate 3 and Mary ate 1 too] ) =>
** 4
:
```

As a last example, here is another way to write sum_nums which makes more use of the stack.

```
define sum_nums( lis );
    vars n;
    0;
    until null( lis ) do
        if isnumber( dest( lis ) -> lis ->> n ) then
            + n;
        endif;
    enduntil;
    enddefine;

:
: sum_nums( [Sid ate 2, John ate 3 and Mary ate 1 too] ) =>
** 6
:
```

Stack operations take place during assignment and during the normal use of infix operators. The stack also provides the mechanism for passing arguments (& results) to (& from) functions.

argtest is a function which takes 3 arguments & displays their values.

```
define argtest( arg1, arg2, arg3 );
    [arg1 = ^arg1, arg2 = ^arg2, arg3 = ^arg3] =>
    enddefine;
:
: argtest( "A", "B", "C" );
** [arg1 = A , arg2 = B , arg3 = C]
:
```

POP is flexible about the way we provide arguments for argtest (or any other funtion). An expression like FN(P, Q, R) generates the code to stack the value of P, stack the value of Q, stack the value of R, then call FN. When FN is called (assuming it has been defined with 3 arguments) the values of P, Q & R will be unstacked & bound to local argument variables.

If FN is called with extra or missing arguments the POP system will not automatically generate a mishap. It will try to continue with its operation. This allows careless programmers to screw up their programs in new & exciting ways. It allows the rest of us to write functions with variable numbers of arguments and sometimes find neater ways to layout complex code.

Here is an example of calling argtest with some/all of its aguments pre-stacked:

```
:
: "A"; "B"; "C"; argtest();
** [arg1 = A , arg2 = B , arg3 = C]
:
: "A"; "B"; argtest( "C" );
** [arg1 = A , arg2 = B , arg3 = C]
:
: "A"; argtest( "B", "C" );
** [arg1 = A , arg2 = B , arg3 = C]
:
```

It is possible to call a function with all of its arguments *pre-stacked* (or a function which takes no arguments) using a different syntax. I will introduce it here & use it later in these notes. The syntax is:

```
.function-name;
```

```
:
: "A"; "B"; "C"; argtest();
** [arg1 = A , arg2 = B , arg3 = C]
:
: "A", "B", "C" .argtest;
** [arg1 = A , arg2 = B , arg3 = C]
:
```

To finish off this section on the stack I'll look at a function, *build*, which takes a variable number of arguments, it is not a very realistic example as there are easier ways to achieve its results but it is simple. It builds a list from items on the stack. It continues removing items from the stack & adding them to its list until it reaches a special symbol. I have used **termin** as the special symbol. In POP **termin** is used as the standard terminating symbol in various situations including end-of-file. In the example it is the *caller's* responsibility to ensure **termin** is placed on the stack. The *caller* is the function/program which calls the function *build*.

```
define build() -> lis;
    /*  builds a list from values on the user stack,
        stops after lifting <termin> from the stack
    */
    vars x;
    nil -> lis;
    until (->> x) = termin do
        x :: lis -> lis;
    enduntil;
    enddefine;

:
: termin;
: for i from 1 to 10 do
:   i;
: endfor;
: build() -> l;
: l =>
** [1 2 3 4 5 6 7 8 9 10]
:
```

Section 3

RECORDS

Record structures are defined using **defclass**, the *field names* given in the **defclass** definition are formed into accessing and updating functions for records. The records themselves (as opposed to the record structure) are constructed from a function formed by the **defclass** statement. The name of this function is formed by appending the record name onto the word **cons**. The syntax for **defclass** is:

defclass <record-name> { ...field-names... };

eg:

```
defclass student    {   name,
        course,
        modules,
        grade
    };
:
: consstudent( "Bill", "computing",
:              [POP11 hardware networks], "merit" ) -> s1;
:
: name(s1) =>
** Bill
:
: hd( modules(s1) ) =>
** POP11
:
:consstudent( "Sue", "Artificial_Intelligence",
:             [LISP Cognition NeuralNets], "pass" ) -> s2;
:
: "merit" -> grade( s2 );
: grade( s2 ) =>
** merit
:
: "Learning" :: modules( s2 ) -> modules( s2 );
: modules( s2 ) =>
** [Learning LISP Cognition NeuralNets]
:
```

In the section about the stack I briefly introduced the idea of using the *.function-name* syntax for calling a function when its arguments had been prestacked (ie: in postfix form). Since the accessors for fields of records are functions we can use this dot notation with them. Many programmers like using this notation with records. It can be used for updating fields as well as accessing them.

```
: s1.course =>
** computing
:
: s1.grade =>
** merit
:
: "distinction" -> s2.grade;
: s2.grade =>
** distinction
:
: "ExpertSystems" :: s2.modules =>
** [ExpertSystems Learning LISP Cognition NeuralNets]
:
```

One word of warning: record fields (like other variables) can be used to hold function values, there is no problem with this except one of precedence if you write statements which use dot-notation to call functions as well as the conventional syntactic form. The function calling mechanism with the conventional syntax takes precedence over the dot-notation. The next example illustrates this.

```
: defclass unlikely { fn, data };
:
: consunlikely( hd, [a b c d] ) -> u1;
;;; DECLARING VARIABLE u1
:
: u1.fn =>
** <procedure hd>
:
: u1.fn( u1.data ) =>

;;; MISHAP - UNLIKELY NEEDED
;;; INVOLVING:  [a b c d]
;;; DOING    :  fn compile Im runproc popvedcommand compile
```

This problem (like other precedence problems) can be overcome with an extra set of brackets:

```
:
: (u1.fn)(u1.data) =>
** a
:
```

LEXICAL CLOSURES

POP allows you to choose different ways to scope variables. **vars** declares dynamically scoped variables, **lvars** declares lexically scoped variables. If you have never come accross the idea of variable scope before and you find the next few pages confusing then get hold of someone who understands all about scoping & have them to explain it to you.

Briefly, dynamically scoped variables are available to any program block which, at run-time, is called from the block in which those variables are declared. This means that if function F calls function G then all variables dynamically declared in F are available to (and modifiable by) G.

Lexically scoped variables are only in scope to the *textual block* in which they are declared. In other words: if you look at a lump of POP code, a lexical variable is only available in the function it is declared in. Note that because scoping is defined by textual blocks a lexical variable is in scope in the block in which it is declared and also any sub-block that block encloses.

EG:

```
define blah()
    lvars X;
    define spam();
        :
        :
    enddefine;
    :
    :
enddefine;
```

X is lexically scoped in **blah**
spam locally defined in **blah**

X is in scope in **spam**

X is in scope in **blah**

Note: if you do not declare variables as being lexically scoped, POP dynamically scopes them. This includes function arguments and function result variables. In order to lexically scope these you must list them in an lvars statement after the function header.

If you have used languages like Pascal or C you will have used lexically scoped variables, but POP and many functional languages take their use one step further. A function operates within an *environment* that is defined by the data objects it can access & its code. Ignoring the issue of dynamic variables: a function's environment is formed when it is defined. This environment is often known as a *closure* (or a *lexical closure*) - it closes off the operating environment of a function.

When **spam** (above) is defined its closure includes any of its local variables and **X**. If **spam** is assigned to a variable (or returned as a value from **blah**) it is the whole closure that is assigned (or returned) - not just the word "spam".

So what? Look at the this example:

```
define list_producer( L );
    lvars L;
    procedure();
        if null(L) then
            nil;
        else
            dest(L) -> L;
        endif;
    endprocedure;
    enddefine;
```

The function `list_producer` returns as its value the closure for an anonymous procedure. This procedure/function uses dest to return `hd(L)` & perform the assignment `tl(L) -> L`.

The closure for this procedure includes **L**. The procedure is defined and a new closure formed each time `list_producer` is called. Each new closure has its own version of **L**. Notice that when `list_producer` finishes **L** only exists as part of an anonymous procedure's closure.

Here is the closure in operation:

```
:
: vars next = list_producer( [egg spam chips and spam] );
: next() =>
** egg
: next() =>
** spam
: next() =>
** chips
: next() =>
** and
: next() =>
** spam
: next() =>
** []
: next() =>
** []
:
```

This next example shows two closures in use at the same time.

```
:
: vars next1 = list_producer( [A B C] ),
:      next2 = list_producer( [X Y Z] );
:
: next1() =>
** A
: next2() =>
** X
: next1() =>
** B
: next2() =>
** Y
:
```

The next example uses the ideas above to get information from a description of a simple world of blocks. This description is a set of [relation object value] statements. For example, the statements:

```
[isa    b1  cube]
[color  b1  red ]
[on     b1  table]
```

mean 'a red cube is on the table'.

If you're bored of lexical scoping you could look at the list below & draw a picture of the blocks it describes, then color it in. If you want to pull a few ideas together, read on.

```
vars blocks;
    [   [isa    b1  cube]       [isa    b2  wedge]
        [isa    b3  cube]       [isa    b4  wedge]
        [isa    b5  cube]       [isa    b6  wedge]
        [color  b1  red ]       [color  b2  red  ]
        [color  b3  red ]       [color  b4  blue ]
        [color  b5  blue]       [color  b6  blue ]
        [on     b1  table]      [on     b2  table]
        [on     b5  table]
    ] -> blocks;
```

This next example has been approached before. The aim, eventually, is to be able to get a list of blocks fulfilling certain criteria. Using our understanding of closures it is possible to write a general purpose function which will manufacture a function closure to detect specific types of statement.

The next function **entry_type** takes 2 arguments: a relation (isa, color, on etc) & a value and returns a function that takes a statement & returns true if the statement has the same relation & value that were provided as arguments to entry_type (otherwise it returns false).

```
define entry_type( reln, valu );
    lvars reln, valu;
    procedure( entry );
        entry(1) = reln and entry(3) = valu;
    endprocedure;
    enddefine;

:
: vars color_blue = entry_type( "color", "blue" );
:
: color_blue( [color b1 blue] ) =>
** <true>
:
: color_blue( [on b1 table] ) =>
** <false>
:
```

This function can be developed a little further so that instead of returning a pure boolean value it returns the object name instead of true for target statements. Don't be put off by the use of 'and'.

```
define entry_type( reln, valu );
    lvars reln, valu;
    procedure( entry );
        entry(1)=reln and entry(3)=valu and entry(2);
    endprocedure;
    enddefine;

:
: vars color_blue = entry_type( "color", "blue" );
:
: color_blue( [color b4 blue] ) =>
** b4
:
: color_blue( [color b2 red] ) =>
** <false>
:
```

It is now possible to use closures returned from entry_names with REMOVE_IF_NOT to produce sets of object names which satisfy certain criteria.

```
:
: vars on_table = entry_type( "on", "table" );
:
: REMOVE_IF_NOT( blocks, color_blue ) =>
** [b4 b5 b6]
:
: REMOVE_IF_NOT( blocks, on_table ) =>
** [b1 b2 b5]
:
: REMOVE_IF_NOT( blocks, color_blue ) #*
:     REMOVE_IF_NOT( blocks, on_table ) =>
** [b5]
:
```

SHARING CLOSURES

It is possible to define two or more local functions within a single lexical block so that they appear to share parts of their closures. In the next example a function make_stack returns a push function (which pushes a value on a stack) and a pop function (which pops a value off the same stack). The data structure used to implement the stack (called STACK) is a list.

Look at the example carefully & notice the following points:

1. Each time make_stack is called it creates new closures for pop & push.
2. Each time make_stack is called it creates a new STACK variable.
3. Any pair of push & pop functions have access to the same STACK variable.
4. The only way to manipulate a stack is by the push & pop functions, the actual datastructure is *hidden* or *private* since it only exists as part of a closure.

```
define make_stack() -> push -> pop;
    lvars STACK = nil, push, pop;

    (procedure(X);  X :: STACK -> STACK;  endprocedure) -> push;

    (procedure();
        if null( STACK ) then
            false
        else
            dest( STACK ) -> STACK
        endif;
    endprocedure) -> pop;

    enddefine;
```

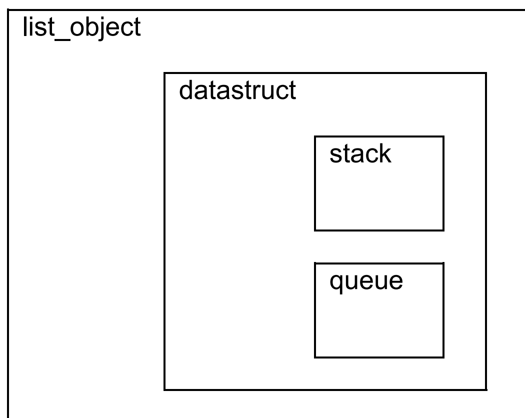
```
:
: vars push1, pop1, push2, pop2;
: make_stack() -> push1 -> pop1;    ;;; make 1st stack
: make_stack() -> push2 -> pop2;    ;;; make 2nd stack
:
: push1( "kipper" );     ;;; put "kipper" & "cod" on 1st stack
: push1( "cod" );
:
: push2( "mango" );
: push2( "peach" );      ;;; put "mango" & "peach" on 2nd stack
:
: pop1() =>
** cod
:
: pop2() =>
** peach
:
: pop1() =>
** kipper
: pop1() =>
** <false>
:
```

The final example that deals with lexical closures builds on the previous example of a stack. It demonstrates how it is possible to build a hierarchy of abstract data objects which have private/hidden parts as well as *public* access functions.

In the example above one function, make_stack, was used as a *builder* and it returned two functions. This time the *builder* is called **make**, it is a general purpose function that returns one value only. For stacks & queues this returned value is a record. It contains all the access functions needed to use a stack or a queue. It is defined like this:

```
defclass Dstruct
{   classname,    ;;; the name of the object "stack" or "queue"
    size,         ;;; Fn to return the number of items it contains
    additem,      ;;; Fn to add a new item (like push for a stack)
    getitem,      ;;; Fn to get an item (pop for a stack)
    print         ;;; Fn to print the contents of stack/queue
};
```

The heirarchy of object types is arranged like this:



Stacks & queues are both types (or *specialisations*) of datastruct & a datastruct is a type/specialisation of list_object.

Make is not included in the diagram. **make** is the function which builds all objects ie: a queue or a stack. **make** takes one argument which is the name of the type of object it is to build. This argument, with a value "queue" or "stack", is the name of the locally defined function called by **make**.

This is how **make** is defined:

```
define make( specialiser ) -> PUBLIC;
    /*  make provides the closure for all objects,
        make only assumes that PRIVATE & PUBLIC parts are needed
    */
    lvars PRIVATE, PUBLIC;

     /*
     all local functions defined here
     */

    /* call specialiser */
    valof(specialiser)();

    /*  all objects have a classname which is the
        name of its specialiser */
    specialiser -> PUBLIC.classname;

    enddefine;
```

Be clear what make does & doesn't do.

1. It sets up a closure for its local functions which contains PRIVATE & PUBLIC parts.
2. It ensures that any PUBLIC parts manufactured by its local functions are returned from make as its result.
3. It calls the *specialiser* function needed to build an object of the required type. This is actually a little complicated. because specialiser is the *name* of a function, rather than the function itself, valof must be used. valof takes an identifier name & returns the value of that identifier. In this case if specialiser = "stack" and make has a local function called stack then valof(specialiser) will return the stack function. In this example specialiser will either be "stack" or "queue" so `valof(specialiser)()` either calls stack() or queue().
4. Make sets the classname field of PUBLIC (which is a record) to be the name of its specialiser.

Make has locallly defined functions for each of the objects in the heirarchy of object types, their declarations are not nested as you might expect but the heirarchy is apparant in the way the functions call each other. EG: stack calls datastruct, datastruct calls list_object.

What follows is a run-down of what each function does, then the complete POP code is given along with a few lines of testing. You will probably need to work through this section more than once to get a clear understanding of it.

You'll notice that the structure for PUBLIC is set up by the most specialised object type while the structure for PRIVATE is set up by the most general object type. This need not always be the case - when you have understood this section you may want to thnk about the advantages & disadvantages of handling PUBLIC & PRIVATE in different ways.

```
define stack();
    initialise PUBLIC to an empty Dstruct record
    call datastruct() to set up default values in PUBLIC

    make the push function for PUBLIC.additem
    make the pop function for PUBLIC.getitem


define queue();
    initialise PUBLIC to an empty Dstruct record
    call datastruct() to set up default values in PUBLIC

    make an add-item-to-queue function for PUBLIC.additem
    make a remove-item function for PUBLIC.getitem
    make a special print function for PUBLIC.print
        NB: this overwrites the default PUBLIC.print set up
        by the datastruct function


define datastruct();
    call list_object() to set up default values in PUBLIC
    make a print function for PUBLIC.print


define list_object();
    PRIVATE is a list for all list_objects,
        initialise it to nil
    make a size function for PUBLIC.size
```

The complete code:

```
define make( specialiser ) -> PUBLIC;
    /*  make provides the closure for all objects,
        make only assumes that PRIVATE & PUBLIC parts are needed
    */
    lvars PRIVATE, PUBLIC;


    /* LOCAL FUNCTION DEFINITIONS */

    define list_object();
        nil -> PRIVATE;      ;;; PRIVATE part of list_object is
                             ;;; initialised to nil
        (procedure(); length(PRIVATE); endprocedure) -> PUBLIC.size;
        enddefine;


    define datastruct();
        /* call super-class */
        list_object();       ;;; datastruct is a list_object
        /* PUBLIC */
        (procedure(); PRIVATE => endprocedure)       -> PUBLIC.print;
        enddefine;


    define stack();
        /*  a stack is a type of datastruct so create
            a datastruct & call the class builder
        */
        consDstruct( undef, undef, undef, undef, undef ) -> PUBLIC;
        datastruct();

        /* add other PUBLIC slots */
        (procedure( x ); x :: PRIVATE -> PRIVATE; endprocedure)
            -> PUBLIC.additem;

        (procedure();
            if null(PRIVATE) then false
            else dest(PRIVATE) -> PRIVATE;
            endif;
        endprocedure) -> PUBLIC.getitem;
        enddefine;
```

```
define queue();
     /* a queue is a datastruct */
     consDstruct( undef, undef, undef, undef, undef ) -> PUBLIC;
     datastruct();

     /* add other PUBLIC slots */
     (procedure( x ); PRIVATE <> [% x %] -> PRIVATE;
     endprocedure) -> PUBLIC.additem;

     (procedure();
         if null(PRIVATE) then false
         else dest(PRIVATE) -> PRIVATE;
         endif;
     endprocedure) -> PUBLIC.getitem;

     (procedure(); 'front => '.pr; PRIVATE => endprocedure)
         -> PUBLIC.print;    ;;; overwrites class definition
     enddefine;

 /* END OF LOCAL FUNCTION DEFINITIONS */


 /* call specialiser */
 valof(specialiser)();

 /*  all objects have a classname which is the
     name of their specialiser
 */
 specialiser -> PUBLIC.classname;

 enddefine;
```

A short test:

```
:
: vars s1 = make( "stack" );
: (s1.additem)("trout");
: (s1.additem)("kipper");
: (s1.additem)("cod");
: (s1.print)();
** [cod kipper trout]
:
: vars q1 = make( "queue" );
: (q1.additem)("Julie");
: (q1.additem)("Joyce");
: (q1.additem)("Jenani");
: (q1.print)();
front => ** [Julie Joyce Jenani]
:
: (s1.getitem)() =>
** cod
: (s1.classname) =>
** stack
: (q1.size)() =>
** 3
:
```

Appendix 1

ASSOCIATION LISTS: ADDITIONAL INFORMATION & UPDATING

infix operator ## with precedence 5
works as an infix call like ASSOC in lisp so:
my_alist ## name
returns the value for the named association in my_alist or <false>
if no such association exists.

see HELP OPERATION, PRECEDENCE

examples:

```
          [    [bus     [color  red]
                        [wheels 6  ]
                        [size   big]
               ]
               [car     [color  blue]
                        [wheels 4   ]
                        [size   med ]
               ]
               [bike    [color  black]
                        [wheels 2   ]
                        [size   small ]
               ]
          ] -> alis;

      :
      : alis ## "bus" =>
      ** [[color red] [wheels 6] [size big]]
      :
      : alis ## "bus" ## "color" =>
      ** [red]
      :
      : "size" -> x;
      :
      : alis ## "bus" ## x =>
      ** [big]
      :
```

```
  Using The Updater
      :
      : alis ==>
      ** [[bus [color red] [wheels 6] [size big]]
          [car [color blue] [wheels 4] [size med]]
          [bike [color black] [wheels 2] [size small]]]
      :
      : [[driver fred] [no 124] ] -> alis ## "bus" ;
      : alis ==>
      ** [[bus [driver fred] [no 124]]
          [car [color blue] [wheels 4] [size med]]
          [bike [color black] [wheels 2] [size small]]]
      :
      : [bert] -> alis ## "bus" ## "driver" ;
      : alis ## "bus" =>
      ** [[driver bert] [no 124]]
      :
```

```
: [[type silly] [size small]] -> alis ## "moped";
: alis ==>
** [[moped [type silly] [size small]]
    [bus [driver bert] [no 124]]
    [car [color blue] [wheels 4] [size med]]
    [bike [color black] [wheels 2] [size small]]]
:
: [yellow] -> alis ## "moped" ## "color";
: [slow] -> alis ## "rollerskate" ;
:
: alis ==>
** [[rollerskate slow]
    [moped [color yellow] [type silly] [size small]]
    [bus [driver bert] [no 124]]
    [car [color blue] [wheels 4] [size med]]
    [bike [color black] [wheels 2] [size small]]]
:
: ;;; using atomic values with conspair
:
: [% "color" | "yellow", "type" | "silly" %]
:       -> alis ## "moped";
:
: alis ==>
** [[rollerskate slow]
    [moped [color|yellow] [type|silly]]
    [bus [driver bert] [no 124]]
    [car [color blue] [wheels 4] [size med]]
    [bike [color black] [wheels 2] [size small]]]
:
: alis ## "moped" ## "color" =>
** yellow
:
: "bert" -> alis ## "bus" ## "driver" ;
: alis ## "bus" =>
** [[driver|bert] [no 124]]
:
```

Appendix 2

FILES {see XSL note}

These notes briefly describe the XSL file handling fns. These fns have superficial similarities with Pascal file readers & writers. One important difference is in the detection of end of file. The end of file marker in POP is **termin** (a special symbol like true or false). The last item read from a file will always be **termin**.

read_open this is a function which takes one argument, a filename (enclosed in ".." or '..'), opens the specified file for reading and returns a stream-specifier (a sort of attachment to a real file). NB: the stream specifier is a function which, if used on its own, will return the next character (as an ascii value) from the opened file.

    read_open(<filename>) -> <stream specifier> ;

write_open this is a function which takes one argument, a filename (enclosed in ".." or '..'), opens the specified file for writing and returns a stream-specifier (a sort of attachment to a real file). If the named file does not exist then a file of that name is created.

    write_open(<filename>) -> <stream specifier> ;

read_item this is a function which takes one argument which must be a stream-specifier returned as a result of a read_open call. read_item returns the next POP object found in the file, this may be an atom or a list. When the end of file is detected read_item returns **termin** and the file is effectively closed - there is no need to close files used for reading.

    read_item(<stream specifier>) -> <POP object> ;

write_item this is a function which takes two arguments the first must be a stream-specifier returned as a result of a write_open call. The 2nd arg is a pop object to be written to the file, this may be an atom or a list.

    write_item(<stream specifier>, <pop object>) ;

writeln_item similar to write_item but also writes trailing newline.

    writeln_item(<stream specifier>, <pop object>) ;

write_close takes a stream specifier as its only arg & closes the specified file.

    write_close(<stream specifier>) ;