TEACH POPCORE
Aaron Sloman September 1997

This is a modified version of the Sussex local file HELP TPOP, by Mike Sharples, and is very close to the summary of Pop-11 in the book Computers and Thought edited by Sharples et al. However, it is more up to date, and reflects local extensions at Birmingham.

This file lists a powerful subset of the words and constructs of Poplog Pop-11 which together are sufficient for a great many interesting programs.

Those items not enclosed in angle brackets `<...>` are Pop-11 reserved words (i.e. they have a special meaning to Pop-11). A relevant help or teach file is indicated by an asterisk,
e.g. * MATCHES. To read the file place the cursor on the asterisk and type `ESC h` or else do `ENTER help matches` Sometimes the TEACH command will produce a different file, e.g. `ENTER teach matches`

CONTENTS

```
ITEM        DESCRIPTION AND HELP FILE           EXAMPLE
----        -------------------------           -------
```

Basic data types

```
<Word>      A letter followed by a series of    "cat"
            letters or digits (including the    "a_long_word"
            underscore). It may also be         "M1"
            a series of signs such as $         "#$#$#$#"
            A word is put in double quotes,
            except within a list or vector
            expression. *WORDS
```

```
<String>     Can contain arbitrary characters.    'A funny %$%### string'
             Constructed using single quotes.
             Can contain "special" characters, E.g.
             \n (newline), \t (tab) *STRINGS      '\ta tab\nand a newline'

<Number>     One or more digits, with an          55
             optional decimal point. *NUMBERS     3.14159
                                                  4.25e10

<List>       A series of text items, such as      [a b c d]
             words, numbers, other lists, or      [1 Acacia Avenue]
             strings, within square brackets.     ['a string' 33 cat]
             *LISTS                               [[CAT CHAT][DOG CHIEN]]

[% ... %]    Decorated list brackets can be       [% for x from 1 to 5 do
             used with enclosed Pop-11 commands          x
             to make a list. *PERCENT                endfor %] =>
                                                  ** [1 2 3 4 5]

<vector>     Like a list, but cannot be extended  {a four word vector}
             and takes up less space.

{% ... %}    Like decorated list brackets can     {% 3+3, 99*9 %} =>
             contain Pop-11 commands.             ** {6 891}

<record>     A kind of structure with a specific  recordclass triple
             number of components and particular    first second third;
             procedures for accessing and         constriple(1, 2, 3) =>
             updating them, etc. *RECORDCLASS     ** <triple 1 2 3>
             (Or see *defclass)
```

Comments

```
;;;          Begins a comment (text that will     ;;; This is a comment.
             be ignored by Pop-11). The comment
             ends at the end of the line. *COMMENT

/* ... */    An alternative form of comment.      /* comments can go
                                                     over several lines */
```

Variable declarations

```
vars         Used to declare local or global      vars x, y, z;
             non-lexical variables. *VARS         vars num = 10;
             (Don't use vars for local
             variables except in a <pattern>).
```

lvars	Used to declare local or global lexical variables *LVARS	`lvars x, y, z;`
		`lvars num = 99;`

Miscellaneous

;	Semi colon terminates commands. I.e. separates imperatives. It is a separator.	`vars a; 100 -> a;`

<undef>	A type of object that is the default value for a variable that has been declared, but not had a value assigned to it. REF * IDENT/'Undef Records'	`vars xxx;`
		`xxx=>`
		`** <undef xxx>`

undef	This special constant refers to an item which is often used as the default value for components of a new structure (e.g. an <array>).	`undef =>`
		`** undef`

Printing facilities

=>	Print arrow. *PRINTARROW	`3 + 4 =>`
		`** 7`

==>	Pretty print arrow (prints a long list or vector tidily). *PRINTARROW	

pr	Prints an item (word, string, list, vector, etc. without "**" or newline.	`pr(list);`
		`pr('The cat');`

ppr	Like pr, but (a) prints lists minus any list brackets, and (b) prints a space after each item. *PPR	`ppr([[the][cat]]);`
		`the cat`

spr	Like pr, but prints a following space *SPR	`spr("a");spr("b");`
		`a b`

npr	Like pr, but prints a newline after each item. * NPR, *PRINTING	`npr("a");npr("b");`
		`a`
		`b`

printf	For more sophisticated printing see *PRINTF	`printf(`
		`  '%p plus %p gives %p',`
		`        [2 3 5]);`
		`2 plus 3 gives 5`

Assignments

`->`	Assignment arrow. Assigns a value to a variable. TEACH * STACK Also invokes updaters, and is used in defining procedures with output locals. See below.	`vars a;` `100 -> a;` `33 -> hd(list);` Compare: `vars a = 100;`
`->>`	Like assignment arrow, but first duplicates object on stack, so that e.g. it can be assigned twice.	`hd(x) ->> a -> b;`

Inserting values in lists and vectors (using "^" and "^^")

`^`	Includes the value of an expression in a list or vector expression. *ARROW	`vars animal = "cat";` `[the ^animal sat]=>` `** [the cat sat]`
`^^`	Includes the elements of a list inside another list. *ARROW	`vars beasts = [cat pig];` `[the ^^beasts sat] =>` `** [the cat pig sat]`
`^ ^^`	NOTE: these also work for vectors	

Defining procedures

<Procedure>	A 'package' of Pop-11 commands, usually with a name. May have an updater *PROCEDURES *DEFINE Some procedures are built-in some user-defined.	hd, sqrt, maplist, *, -, subscr, etc. are all built-in procedures.
define enddefine	Start and end of a procedure definition *DEFINE	`define perim(width,height);` `return(2*width + 2*height)` `enddefine;`
return	Terminates execution of the current procedure, and returns to whatever invoked it. Analogous to "goto enddefine". Items in brackets after return are left on the stack. *RETURN	`define first_and_last(list);` `return(hd(list),` `last(list))` `enddefine;`
->	Indicates an 'output local' in a procedure header line. An alternative to 'return' as a way of specifying the result of a procedure call. *DEFINE, *STACK	`define perim2(w,h)->result;` `2*w + 2*h -> result;` `enddefine;`

Miscellaneous 2

readline()	A Pop-11 procedure that prints a ? and then waits for input from the terminal. Any words, numbers or strings typed on the line after the ? are returned in a list. *READLINE	`readline() -> input_words;`
date()	A procedure that returns a list giving the current time and date. *DATE	`date()=>` `** [18 Sep 1985 11 47 16]`
length(<item>)	A procedure that returns the length of an item. *LENGTH The length of a item is the number of components it contains.	`length([the cat sat])=>` `** 3` `length("iguana")=>` `** 6`

`<Subscript>`	An element can be picked from a list by giving its position in brackets after the name *LISTSUMMARY	`vars sentence animal;`
		`[the cat sat] -> sentence;`
		`sentence(2) -> animal;`

`oneof(<list>)`	Returns an element picked at random from a list. *ONEOF	`vars throw =`
		`   oneof([1 2 3 4 5 6]);`

Arithmetic operators

`+`	Adds one number to another.	`width+height->half_perim;`
`*`	Multiplies two numbers.	`3.14159*d -> circum;`

`/`	Divides one number by another. Warning: dividing one integer by another can give a "ratio" which may print as, e.g. 3_/4	`total/items->average;`
		
		`10/5, 3/4 =>`
		`** 2 3_/4`

`abs`	When applied to a positive or negative number returns its absolute value (always positive) *ABS	`abs(-10) =>`
		`** 10`

`pop_pr_ratios`	This Pop-11 variable controls how ratios are printed. If made false it makes ratios print as decimals.	`false -> pop_pr_ratios;`
		
		`10/5, 3/4 =>`
		`** 2 0.75`

`//`	Divides one integer by another to get dividend and remainder,	`10//3`
		` -> (remainder,dividend);`

`**`	Raises one number to the power of another.	`2**3 =>`
		`** 8`

`>`	Compares two numbers. The result is true if the first is greater.	`if x > 3 then .... endif`
`>=`	Compares two numbers. The result is true if the first is greater or equal.	

`<`	Compares two numbers. The result is true if the first is smaller.	`4 < 3  =>`
		`** <false>`

<=	Compares two numbers. The result is true if the first is smaller or equal to the second.	
( )	Round brackets have two uses. They can alter the order of evaluation in expressions, or following a variable or expression they can signify procedure invocation. Any arguments to the procedure go in the brackets.	(3+2)*4 => ** 20 perim(45,23) => ** 136

true
false
	These are constants which hold the two special boolean values <true> and <false> used in conditionals and loop termination tests. *BOOLEAN	ȷtrue =>
		** <true>
		ȷfalse =>
		ȷ** <false>

=	Tests whether two items are equal *EQUAL
	It can also be used to initalise a variable;
		if a = 100 then ...
		
		vars x = [1 2 3];

==	Tests whether items are identical	if a == [cat] then ...
/=	Tests whether two items are unequal. (Looks inside structures) * EQUAL	a /= b
/==	Tests whether two items are not identical. (Does not look inside structures)	a /== "cat"

Logical connectives

(E.g. for use in conditionals)

and	Forms the 'conjunction' of two boolean expressions. *AND	if x > 0 and x < 100 then
or	Forms the 'disjunction' of two boolean expressions. *OR	word="cat" or word="puss"
not	Negates a boolean expression. *NOT	not(list matches [== cat ==])

if	Marks the start of an 'if' conditional. *IF	if english == "cat" then
		   "chat"=>
		endif;

then	Ends the condition part of an 'if' conditional. *THEN (Also used with "unless")	

elseif	Begins a second (or subsequent) condition in an 'if' statement. *ELSEIF	if english == "cat" then
		  "chat" =>
		elseif english == "dog" then
		  "chien" =>

else	Marks the beginning of the "default" course of action in a conditional. *ELSE	else
		  [I dont know] =>
		endif;

endif	Marks the end of a conditional. *ENDIF	

Variable formats for the matcher

matches	Compares a list with a pattern. It returns true if they match, false otherwise. It will also "bind" variables in the pattern, if there are any. *MATCHES	vars sentence;
		[the cat sat] -> sentence;
		sentence matches [= cat =] =>
		 ** <true>

=	Matches one item inside a list pattern.	mylist matches [= cat sat]
==	Matches zero or more items inside a pattern.	mylist matches [== cat ==]
?<variable>	Matches one item inside a list pattern and makes that the value of the variable. *MATCHES	mylist matches [?first ==]
??<variable>	Matches zero or more items within a list pattern and makes the list of matched items the value of the variable. *MATCHES	alist matches [?first ??rest] => ** <true>
!	Use in front of a pattern to make the variables lvars	mylist matches ![?first ??rest] =>
database	A Pop-11 variable whose value is the database, a list of lists, used with add, remove, present, etc. *DATABASE	database ==>

add(<list>)	Puts an item into the database. *ADD	add([john loves mary]);
remove(<pattern>)	Removes the first item matching the pattern from the database. *REMOVE	remove([john loves =]);
flush(<pattern>)	Removes all items matching the pattern from the database. *FLUSH	flush([== loves ==]);

present(<pattern>)	Searches the database for an item matching the database and returns true if it is found, false otherwise. Binds variables in the pattern. *PRESENT	if present([?x loves mary])
		then
		   x=>
		endif;

allpresent(<list of patterns>)	Searches the database for items that consistently match all the patterns, and returns true if this succeeds and false otherwise Binds variables in the pattern.
	*ALLPRESENT
		if allpresent(
		  [[?x loves ?y]
		      [?y loves ?z]])
		then
		   [Triangle ^x ^y ^z] =>
		endif;

it	A variable that is set by 'add', 'remove', 'present' and 'foreach'. Its value is the last item found in the database. *IT	if present([?x loves mary])
		then
		    it=>
		endif;

Looping expressions

repeat	Marks the start of a repeat loop.
	*REPEAT
		repeat
		  readline()->line;
		  quitif(line /== []);

endrepeat	Marks the end of a repeat loop. *ENDREPEAT	endrepeat;

times	Indicates the number of times a repeat loop is to be repeated (If it is omitted then looping is forever,unless halted by quitif).
	*TIMES
		repeat 4 times;
		    "."=>
		endrepeat;

```
quitif(<expression>)                          vars n = 2;
            If the expression is true then    repeat;
            quit the loop. This example and     quitif(n > 1000);
            the one using the while loop        n =>
            below are equivalent (ie they       n*n -> n;
            give the same result). *QUITIF    endrepeat;

while       Marks the start of a while loop.  vars n = 2;
            *WHILE                            while n <= 1000 do
                                                 n =>
do          Ends the condition part of a         n*n -> n;
            'while', 'for', or 'foreach' loop. endwhile;
            *DO

endwhile    Marks the end of a while loop.
            *ENDWHILE

for         Marks the start of a for loop.    for x in [paris london] do
            *FOR                                [^x is a city]=>
                                              endfor;
endfor      Marks the end of a for loop.
            *ENDFOR
            Note: there are many different forms
            of for ... endfor loops.
            See *LOOPS, *FOR.

foreach     Marks the start of a foreach loop, vars x y;
            which matches a pattern against   foreach [?x loves ?y] do
            each item in the database. *FOREACH   it=>
                                              endforeach;
endforeach  Marks the end of a foreach.
            *FOREACH

forevery    Like foreach, but takes a list of forevery
            patterns and tries all possible      [[?x ison ?y]
            ways of matching them all             [?y ison ?z]]
            consistently with items in the    do
            database. *FOREVERY                   them =>
                                                  [^x is above ^z] =>
endforevery Syntax word used at the end of    endforevery;
            a "foreach" loop.
```

Arrays

```
<array>     A compound data object with N dimensions
            whose components can be accessed or
            updated using N numerical subscripts.
            *ARRAYS
```

`newarray`	The simplest procedure to create a Pop-11 array. * NEWARRAY	`vars ten_by_seven = newarray( [1 10 -3 3]);`
`boundslist`	When applied to an array returns a list containing for each dimension the upper and lower bounds.	`boundslist( ten_by_seven) =>` `** [1 10 -3 3]`

Tracing utilities

`trace <names of procedures>`	A command that alters procedures so they print out helpful information. (NB. You can trace built-in procedures like 'hd' and 'tl'). *TRACE	`trace add first_and_last;`
`untrace <names of procedures>`	A command that switches off tracing of the named procedures. *TRACE	`untrace add first_and_last;`
`untraceall`	Switches off any traces.*UNTRACEALL	`untraceall;`

See also *INSPECT and *DEBUGGER

FURTHER READING

The **Pop-11 Primer**, by A.Sloman, is available online as TEACH PRIMER and also available in hard copy from the School of Computer Science Library.

```
TEACH * FACES,  * GSTART,  * USEFULKEYS

TEACH *LISTS, *LISTSUMMARY,

TEACH *BOXES *POPSUMMARY, *DEFINE, *STACK, *VARS

TEACH * DATABASE, * FOREACH

HELP * WORDS, *LISTS, *MATH, *LOOPS, *CONTROL, *ARRAYS, *STRINGS

HELP * MATCHES, *PRINT, *TRACE, *RECURSION

TEACH * RECURSION, * SETS, * SETS2, * FUNCTIONAL.STYLE

M. Sharples, et al.
    Computers and Thought,
    MIT Press, 1989
        (This is an introduction to cognitive science using
        Pop-11 programming examples as illustrations.)

James Anderson(ed)
    Pop-11 Comes of Age
    Ellis Horwood, 1989
        (A collection of papers on the history of dialects of Pop,
        the features and benefits of the language, and some
        applications using Pop-11.)

Chris Thornton & Benedict du Boulay (1992)
     Artificial Intelligence Through Search
     Kluwer Academic (Paperback version Intellect Books)
        (An introduction to AI using Pop-11 and Prolog. A good
        way to learn Prolog if you know Pop-11 or vice versa.)
```

WARNING: books published before 1995 are likely to have out of date information about Pop-11, though the core ideas are unchanged.

In the Poplog system there is a large collection of **REF** files giving definitive information about Pop-11. These files are mostly useful for experts, but occasionally you'll find that information you need is available nowhere else.

The pop-forum email list and comp.lang.pop internet news group are also useful sources of information. There is a lot of pop-11 material available by ftp from the Birmingham Poplog directory

http://www.cs.bham.ac.uk/research/projects/poplog/