Poplar Language Manual

Jim Morris
Eric Schmidt
November 1, 1978
(Revised: December 1980, version 0.3)

Poplar is an experimental language for text manipulation. We hope to apply it to a variety of tasks commonly found in office environments. Ultimately, we hope to develop a language suitable for use by Xerox customers, but the present language will be too difficult for a non-programmer to assimilate. Its goal is simply to explore the range of functionality such a language might have. It is based on ideas from LISP, SNOBOL, APL and the UNIX operating system. Certain of your favorite features have been left out or given strange implementations in order to encourage exploration of different programming styles. Read this, try the implementation, and give us some feedback. Example programs, successful or not, will be greatly appreciated.

The implementation is more of a sketch than a finished picture: it represents three months effort by one and one-half of us, so it has not been possible to provide all the things one might think of or eliminate all the bugs one might find.

This manual is composed of several sections:

It has some appendices:

C: Syntax Equations for Poplar

D: Getting Started

Several people have made very helpful suggestions to improve this document and the language: Doug Clark, Gene McDaniel, Paul McJones, Tom Moran, Alan Perlis, Dana Scott.

Expressions

Poplar is an *expression*-oriented language (rather than statement-oriented or command-oriented). Poplar programs consist of expressions. The implementation is an *evaluator* which reduces an expression to a simple form, called its *value*. Throughout this manual we shall write things like

```
1+2 = 3
```

which suggests that the evaluator will transform `1+2` into `3`.

In general, an expression is a *constant* like `2` or `"abc"`, a *variable* like `x` or a compound expression consisting of one or two sub-expressions and an *operator*, like +. An operator may be *binary*, in which case it separates two subexpressions; it may be *unary prefix*, in which case it precedes a single subexpression; it may be *unary postfix*, in which case it follows a single subexpression. Subexpressions are often enclosed in parentheses `()`, brackets `[]` or braces `{}`. For example

```
[(((~"abc") !), (1 + 2)]
```

is an expression composed of many nested subexpressions. Often, parentheses can be omitted. The above expression is the same as

```
[(~"abc")!, 1 + 2]
```

The implicit parenthesization rules are based on the concept of *precedence*. We say that an operator, !, has higher precedence than another one, +, if it is performed first on an operand placed between them, i.e.

```
x+y! = x+(y!)
```

In general, unary postfix operators have the highest precedence, followed by unary prefix, then binary operators. If two adjacent operators have equal precedence the one on the left is performed first. Thus

```
x+~y!+y=(x+(~(y!)))+y
```

There are exceptions to these rules. They will be noted when the effected operators are introduced. A complete set of rules for expression formation is given in Appendix C.

Strings and Lists

There are three types of values in Poplar: Strings, Lists, and Functions. Functions may be *primitive* (built-in), *programmer-defined*, or *patterns*. There is also a special value fail.

Any sequence of characters enclosed in quotes is a *string*. For example,

```
"I am a string"
""
```

are two strings. The second is called the *empty string*.

Numbers are also strings, but need not be quoted; e.g. 123 = "123".

Special, two-character combinations can be used inside strings to represent characters which are inconvenient or impossible to type:

^" is a quote mark
^ is a space (There is a space after that "^")
^^ is just ^

In general for any upper-case letter X from A-Z, ^X is the ASCII character control-X. In particular,

^M is a RETURN
^I is a TAB
^Z is control-Z

Finally, ^nnn where nnn is precisely three digits is considered a single character with that octal ASCII code.

A *list* is a sequence of things denoted using brackets and commas. For example,

```
["abc", "xyz"]
[16]
[]
```

are lists, the last being the empty list. Lists can be components of lists, e.g.

```
["roster", ["jim", "417 Smith St.", "555-7821"], ["fred", "625 B St.", "555-9021"]]
```

is a list of three items, two of which are lists of three items. Other kinds of values discussed later (functions, patterns) can also be list elements.

Considered as an operator, a comma, has the lowest possible precedence, so it is never necessary to enclose a list element in parentheses.

Function Application

The operator / (*application*) means apply the function that follows it to the preceding value.

There are many primitive functions. For example, the function length may be applied to strings or lists to produce the number of characters in the string or items in the list.

```
"abc"/length = 3
""/length = 0
["abc", "c"]/length = 2
[]/length = 0
```

Functions with two or more inputs take them in a list.

The function conc will combine a list of strings into one.

```
["abc", "def"]/conc = "abcdef"
[123, 678]/conc = 123678
["abc", ""]/conc = "abc"
```

The functions plus, minus, times, and divide may be applied to pairs of numbers.

```
[1,3]/plus = 4
[1,"a"]/plus is an error
[4,6]/minus = -2
[3, -8]/times = -24
[7, 3]/divide = [2, 1]
```

The value of a divide function application is a quotient and remainder.

Since they are used so frequently, there are shorter notations for conc, plus and minus.
Concatenation is designated simply by juxtaposition.

```
"abc" "def" = "abcdef"
```

Plus and minus are designated by + and -.

```
123+4 = 127
5-10 = -5
```

The various components of a list may be designated by applying integers to the list, as if the integer were a function. The numbering starts with 1.

```
["ab", "c", "d"]/1 = "ab"
["ab", "c", "d"]/3 = "d"
["ab", "c", "d"]/0 is an error
["ab", "c", "d"]/4 is an error
```

Applying a negative number to a list results in a list shortened by removing that number of elements from the beginning.

```
["ab", "c", "d"]/-1 = ["c", "d"]
["ab", "c", "d"]/-3 = []
["ab", "c", "d"]/-4 is an error
```

Two lists may be concatenated by placing two commas between them.

```
["abc", "def"] ,, [123, 456] = ["abc", "def", 123, 456]
[123, 456] ,, [0] = [123, 456, 0]
["a", "b"] ,, [] = ["a", "b"]
```

The expression x--y will generate a list of numbers starting with x and ending with y.

```
1--7 = [1, 2, 3, 4, 5, 6, 7]
3--3 = [3]
4--2 = [4, 3, 2]
```

The function sort rearranges a list in ascending alphabetical order.

```
["beta", "zero", "alpha"]/sort = ["alpha", "beta", "zero"]
```

If an element of the list is itself a list, sort assumes the first component of the list is a string and uses that string in deciding where the list goes.

```
[["fred", 20], "al", ["jane", 3]]/sort = ["al", ["fred", 20],  ["jane", 3]]
```

There is a complete description of Poplar primitive functions in Appendix B.

Conditional Expressions

Certain primitive functions islist, isnull, and isstring, return the special value fail if their input is not as they describe it and return the input otherwise.

```
[2, 3]/islist = [2, 3]
"2,3"/islist = fail
[]/isnull = []
[3, 4]/isnull = fail
"a"/isstring = "a"
[5]/isstring = fail
```

The operation of *matching*, described fully below, can also return fail. This simplest case of matching is a test for string equality written as string1/{string2}.

```
"abc"/{"abc"} = "abc"
"abc"/{"a"} = fail
```

The value fail may be used to select between different values. The relevant operators for doing this are | and >.

The operator | (*otherwise*) has the following behavior.

```
V | F = V      if V is not fail (F is not evaluated at all.)
fail | F = F
```

Thus

```
x/isstring  | "x is not a string"
```

will be x if x is a string, but will be "x is not a string" otherwise. Operationally, E | F means: Evaluate E; if the value, V, is not fail, forget about F and take V value of the entire expression. Otherwise, evaluate F and take its value (fail or not) as the value of the whole expression. The otherwise operator has a lower precedence than most other operators so that

```
x/y | t/u = (x/y) | (t/u)
```

The operator > (*then)* has the following behavior

```
fail > F = fail          (F is not evaluated at all.)
V > F = F                if V is not fail
```

Thus

```
x/{"abc"} > "xyz"  | "def"
```

will be "xyz" if x is "abc", but will be "def" otherwise. Operationally, E>F means: Evaluate E; if the value is not fail, forget the value of E and evaluate F for the value of the whole expression. If the value of E is fail return fail as the value of the whole expression. The then operator has a lower precedence than most other operators (save |), so that

```
x>y | t/v > l/k = (x>y) | ((t/v) > (l/k))
```

The expression

```
P > E | F
```

is almost, but not quite, equivalent to the Algol **if** P **then** E **else** F. It differs when E=fail.

The *not* operator ~ maps fail into the empty string and everything else into fail

```
~ fail = ""
~ "" = fail
~ "abc" = fail
```

Iteration on Lists

All the binary string operations may be applied to lists. If one of the operands is not a list then it is combined with every element of the other list.

```
4 + [-2, 3, 8] = [2, 7, 12]

"foo." ["bcd", "mesa"] = ["foo.bcd", "foo.mesa"]

"<fred>" ["form", "eval", "comp"] ".mesa"
        = ["<fred>form.mesa", "<fred>eval.mesa", "<fred>comp.mesa"]
```

If both operands are lists then they must be the same length and the operation is applied element by element.

```
["abc", "def"] ["xyz", "123"] = ["abcxyz", "def123"]
[5, 6] + [7, 8] = [12, 14]
[5, 6] + [7, 8, 14] is an error
```

If a list follows an application operator, the result is the same as applying each element of the list to the preceding value and forming a list of the results.

```
[12, 7]/[length, conc, plus] = [2, 127, 19]
["a", "b", "c"]/[2, 1] = ["b", "a"]
[10, 20, 30, 40, 50, 60]/(2--5) = [20, 30, 40, 50]
```

The operator // (*maplist*) will apply the following function to every member of the preceding list and create a new list of the results.

```
["a", "bcd", ""]//length = [1, 3, 0]
[[3,4], [6,9], [-4,7]]//times = [12, 54, -28]
[["a","b"], ["c", "d"], ["x", "Y"]]//2 = ["b", "d", "Y"]
[]//f = [] for any f
```

If the value of an application is fail it is omitted from the result list.

```
["a", ["x"], "b", ["t", "c"]]//isstring = ["a", "b"]
```

It is often useful to process all the items in a list while accumulating some information. This can be accomplished using the operator /// (*gobble*).

```
[x1, ... , xn] /// f.
```

applies f to pairs of items. It starts with x1 and x2 to produce y1; then it combines y1 with x3 and so on.

```
[1, 4, 9, 20]///plus = 1+4+9+20 = 34
[2, 4, 8]///minus = 2-4-8 = -10
["The ", "quick ", "brown ", "fox "]///conc = "The quick brown fox "
```

More generally,

```
[x]///f = x
[x1, x2, x3, x4, ...] /// f = [[x1, x2]/f, x3, x4, ...]///f
[]///f is an error
```

Variables, Assignment, and Sequencing

A *variable* is either a single letter or a sequence of letters including at least one capital letter. Thus variables can always be distinguished from special Poplar names, like conc. One can assign values to variables with the assignment operator _.

```
x _ "A long string I would rather not type repeatedly"
BlankLine _"^M^M"
```

Subsequently evaluated expressions containing the variable will use the value in place of the variable. The value of x _ e is e.

```
x BlankLine x =

    "A long string I would rather not type repeatedly

    A long string I would rather not type repeatedly"
```

Expressions may be evaluated solely for their side effects; in which case the are called *statements*. When this the case it is desirable to combine them into sequences and ignore the values they produce. The semicolon is used to separate such items.

```
(x _ 1; y _ 3; x+y) = 4
```

Thus the value of such an expression is the value of the last sub-expression.

The semicolon and _ have precedence lower than |.

```
x _ y _ a | b; z _ a-y = (x _ (y _ (a | b))); (z _ (a-y))
```

Programmer-defined functions

A function can be described by writing a variable followed by a colon followed by any expression involving the variable. The expression following the colon is called the *body* of the function.

```
x: x "O" x
```

The effect of applying a such a function to a value is to substitute the value for the corresponding names.

```
"W" / x: x "O" x
```

yields

```
"W" "O" "W"
```

which eventually yields

```
"WOW"
```

A list of variables may appear before the colon. In that case the function must be applied to a list of values with the same length. Each value is substituted for the corresponding variable.

It is often convenient (or perhaps only amusing) to use an in-line function to name values, rather than an assignment statement. For example, the previous sequence

```
x _ "A long string I would rather not type repeatedly";
BlankLine _"^M^M";
x BlankLine x
```

could be re-written

```
"A long string I would rather not type repeatedly" / x:
"^M^M" / BlankLine:
x BlankLine x
```

The operators // and /// may be used with programmer-defined functions

```
["a", "bc", "ttt"] // (x:  x "@" x) = ["a@a", "bc@bc", "ttt@ttt"]

[[1, 2], [3, 8], [5, 2]] // ([x, y]: [x+y, x-y]/times) = [-3, -55, 21]

["a", "b", "c", "d]///([x, y]: y x) = "dcba"
```

["a"]/([x,y]: x y y) is an error because the length of the lists differ.

A function may be assigned to a variable and then that variable may be referenced like a primitive function.

```
F _ (x: x x x); "a"/F = "aaa"

(x: x x x)/F: "a"/F = "aaa"
```

The symbol : considered as an operator has unique precedence properties: It has the highest precedence of all when viewed from the left, and the lowest precedence of all when viewed from the

right. This is so the variables introduced on its left have a scope limited only by closing parentheses.

```
a+b/x: t _ x/conc; x+1   = (a+b)/(x: (t _ x/conc; x+1))

Successor _ x: x+1; Predecessor _ x: x-1;
```

is, surprisingly, the same as

```
Successor _ (x: x+1; Predecessor _ (x: x-1));
```

Thus one will usually enclose programmer defined functions by parentheses.

Patterns and Matching

Patterns are functions used to analyze strings. In general, a pattern is denoted by any expression enclosed in braces {}. Applying a pattern to a string is called *matching*. The result is equal to the string if the match succeeds; otherwise it is equal to fail.

Any string can become a pattern. The match succeeds if the strings are equal. Upper- and lower-case characters are always different.

```
"abc"/{"abc"} = "abc"
"abc"/{"aBc"} = fail
```

Patterns may be combined with the operator | to form a new pattern which matches either the first or second component.

```
"ab"/ {"ab"|"c"} = "ab"
"cd"/ {"a"|"cd"} = "cd"
"cd"/ {"a"|"b"|"d"} = fail
```

It is permissible, but not required, to put braces around sub-patterns in patterns, e.g.

```
{{"a"|{"b"}}|"d"}
```

is a legal pattern.

Patterns may be concatenated to form a new pattern that matches the concatenation of any strings that match the individual patterns.

```
"ad"/ {("a"|"c") ("b" | "d")} = "ad"
"xcd"/  {"x" ("a"|"cd")} = "xcd"
```

The pattern

```
P!
```

matches an arbitrarily long sequence of one or more P's. For example,

```
"aaa" /  {"a"!} = "aaa"
```

The pattern

```
P?
```

matches an optional P; if the match succeeds, fine; if not, fine too.

```
"abc"/ {"a" "b"? "c"} = "abc"
"ac"/ {"a" "b"? "c"} = "ac"
```

The idiom P!? can be used to indicate zero or more repetitions of P.

The pattern # (*wild card*) matches any single character.

```
"abc"/{"a" # "c"} = "abc"
```

The pattern ... (*ellipsis*) matches any sequence of zero or more characters whatsoever. The matcher endeavors to make the substring that it matches as short as possible, subject to the item following the ... matching successfully. A ... at the end of a pattern matches everything to the end of the string.

```
"abc" / {... "c"} = "abc"
"abc,def,ghi,bbb," / {(... ",")!} = "abc,def,ghi,bbb,"
"abcxyzsss" / {... "xyz" ...} = "abcxyzsss"
16/{"-"...} = fail
-16/{"-"...} = -16
```

The operator ~ may occur in patterns. As before, it changes fail into the empty string and anything else into fail.

```
"abc" / {~"abc"} = fail
"abc" / {~"x"} = fail
"abc" / {~"x" "abc"} = "abc"
```

The idiom {~P ...} matches anything which does not begin with a P.

The following are some useful, pre-defined patterns:

```
digit = {0|1|2|3|4|5|6|7|8|9}
integer = {"-"? digit!}
number = {"-"? ("." digit! | (digit!  ("."  digit!?)?))}
smallletter = {"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"}
bigletter = {"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
| "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"}
letter = {smallletter | bigletter}
word = {letter!}
item = {word | number}
thing = {(letter|digit)!}
space = {" " | "^I"} (blank or tab)
```

Two other patterns can be modified by a number

```
blanks 5 = "^ ^ ^ ^ ^ "
len 3 = ###
```

Matching Combined with Evaluation

It is usually desirable to extract more information from the matching process. This can be accomplished by adding structure to patterns with the normal set of operations available for strings, lists, and functions. The strings that match individual components of a pattern are then recombined under control of those operations.

For example, the expression

```
"How many times"/{ [word, " " word, " " word] }
```

evaluates to become

```
["How", " many", " times"]
```

Operationally, the string was broken into five pieces by the pattern: "How", " ", "many", " ", and "times". Then the concatenation of the spaces onto "many" and "times" was performed, and then the three pieces were made into a list.

The *deletion* operator * when suffixed to an expression forces it to evaluate to be the empty string unless it is fail. It is useful for discarding portions of a matched string.

```
"How many times"/{[word " "* word*, " "* word)]} = ["How", "times"]

"56,89"/ {number ","* + number} = 145
```

The pattern

```
P,!
```

with a comma just before the ! behaves just like P! except that the individual items that match P are combined into a list rather than being re-concatenated into a string.

```
"aaa"/ {"a",!} = ["a", "a", "a"]
"The quick brown fox" / {(word " "?*),!} = ["The", "quick", "brown", "fox"]
```

The application operators (/, //, ///, %) and > are also permitted in patterns, but are treated unlike the other binary operators in that their right operands do not participate in the match, but are combined, as is, with whatever matches the left operand.

The conditional operator > can be used to replace a (non-fail) value with something else.

```
"abc"/{...("b">"x")...}= "axc"

Month _ {"Jan" > 1 | "Feb" > 2 | "Mar" > 3 |  "Apr" > 4 | "May" > 5 | "Jun" >
6 | "Jul" > 7 | "Aug" > 8 | "Sep" > 9 | "Oct" > 10 | "Nov" > 11 | "Dec" > 12}

"Apr"/Month = 4
```

When an application operator appears inside a pattern the function that follows it is applied to whatever matches the subpattern before it.

```
"abc-def-hijk-n-"/{(word "-"* / length),!} = [3, 3, 4, 1]
```

Operationally, this could have happened in two stages. First the matching process yields

```
[("abc" "-"*/length), ("def" "-"* /length), ("hijk" "-"* /length), ("n" "-"*/length)]
```

Then the evaluation process throws away the "-"'s and computes the lengths.

```
"abcd" / {("ab" / x: x x x) ...} = "abababcd"
"23-Jan-78" / {[integer, "-"* word, "-"* integer]
        / [d, m, y] : m " " d ", 19" y}
= "Jan 23, 1978"
```

Using the pattern Month from above,

```
"23-Jan-78" / {[integer, "-"* Month, "-"* integer]
        / [d, m, y] : m "/" d "/" y}
= "1/23/78"
```

The idiom {...(P > S)...} means replace the first occurrence of P with S. (That is: the value of the match will be a new string derived from the old by replacing P with S).

The idiom {(...(P > S))!...} means replace all occurrences of P with S.

The Matching Process

In simple situations the matcher usually does the expected thing. Occasionally, however, it will surprise you by failing to match something you thought it should. That is because the matcher follows a rather simple, left-to-right, matching rule and doesn't usually back up in the string it is trying to match. Specifically, given a pattern like

```
s/{P1 P2}
```

it finds a prefix of s which matches P1, then tries to match P2 against the remainder of the string. If P2 fails to match the remainder the entire match fails. There might be a different way to match P1 against s that consumes more or fewer characters so that P2 *would* match the remainder. Nevertheless, the matcher does not bother trying new ways to match P1 (unless P1 is an ellipsis; see below.).

For example,

```
"abc" / {("a" | "ab") "c"} = fail
```

because the first alternative, "a", was chosen to match the "a" in the subject string, and the matcher did not back up to try the "ab" alternative when "c" fails to match "b". On the other hand

```
"abc" / {("ab" | "a") "c"} = "abc"
```

This suggests that if one alternative is a prefix of another you should put the longer one first.

Another example: the matcher finds the longest sequences it can so

```
"aaabc" / {"a"! "abc"} = fail
```

because the third "a" was used up by the "a"!.

The only situation in which the matcher backs up involves the ellipsis pattern. If, in {P1 P2}, P1 is an ellipsis or ends with an ellipsis the matcher begins by assuming the ellipsis matches the empty string and extends the ellipsis match, one character at a time, until P2 matches the remainder of the string. Thus the ellipsis pattern ... is quite different from the apparently similar pattern #!?. The second one is fairly useless (except for signifying expletives) since it uses up the entire string it is applied to.

```
"aaaab"/{... "b"} = "aaaab"
"aaaab"/{#!? "b"} = fail
```

The use of ellipsis can be surprisingly expensive occasionally. For example,

```
"abcd  def wddf: x"/{... (word ":") ...}
```

will cause the pattern word to match seven different substrings ("abcd", "bcd", "cd", "d", "def", "ef", "f") before coming to rest on "wddf".

General Iteration and Recursion

If you want to do something repeatedly and it doesn't correspond to marching down a list you can use the operator %.

```
E % f
```

applies f to E repeatedly, calling the result the new E, until E is fail, then returns the previous value for E.

```
-10 % (x: x+4/{"-" ...})  = -6 % (x: x+4/{"-" ...})

                          = -2 % (x: x+4/{"-" ...})

                          = -2
```

In general,

```
E % f = E/f > E/f % f | E
```

You can write recursive functions, if you like.

```
Blanks _ (n: n/{0} > "" | "^ " (n-1/Blanks))
```

generates a string of n blanks.

You can also write recursive patterns.

```
AE _ {integer | "(" AE ("+" | "-") AE ")" }
```

matches strings like "(5-(6+2))".

AE suggests a fairly succinct evaluator for fully parenthesized arithmetic expressions

```
Eval _ {integer |
        "("* Eval "+"* +  Eval ")"* |
        "("* Eval "-"* - Eval ")"*};

"(5-(6+2))"/Eval = -3
```

Equality Assertions

Equality assertions are intended to be an aid to program development, documentation, and maintenance. They provide a means of interleaving an example with a program. An assertion is a phrase of the form

```
= E
```

where E is any expression (in practice, a constant). Any programmer defined function may be decorated with equality assertions. One uses *premise assertions* after the input variables and *conclusion assertions* in the body of the function to describe the value of intermediate results.

Consider the function

```
([x, y]: [x,y]/marry//times///plus)
```

We can add assertions to it to produce the following, equivalent program

```
([x,y] := [[1,4,8], [2,-9,3]];
                [x,y]/marry = [[1,2],[4,-9],[8,3]]
                //times///plus = -9)
```

This says: If x happens to be [1,4,8] and y happens to be [2,-9,3] then the value of [x,y]/marry is [[1,2],[4,-9],[8,3]] and the final value is -9.

In general, one may add a conclusion assertion to any expression, as long as it occurs inside a function which has premise assertions for its variables.

Normally, these assertions are regarded as comments, but one can have the evaluator check them, too. Given a function with premise assertions, it substitutes the values for the variables and evaluates the body checking that each conclusion assertion is true.

When a function occurs within a function, things are a little more complicated. The problem is that each conclusion assertion should be checked only once, even if the inner function is used repeatedly. Consider the following function

```
(x:=["abc", "z"]; x // (y: [y/length, y])
                                    //sort//2 = ["z", "abc"])
```

If we want to attach an assertion to the result of the inner function, which should we use: = [3, "abc"] or = [1, "z"]? The answer is "neither"; we invent an independent premise assertion for the inner function.

```
(x:=["abc", "z"]; x // (y="gh": [(y/length), y] = [2, "gh"])
                                    //sort//2 = ["z", "abc"])
```

The checking evaluation proceeds as follows:

1. Start the check of the outer function: Substitute the premise value for x.
2. Start the check of the inner function: Substitute the premise value for y, and check the result of the inner function. The inner function is now checked.
3. Apply the inner function to "abc" and "z".
4. Sort the list, discard the lengths and check the final result.

It might be helpful to a reader to choose the examples for inner functions to correspond to a particular case based upon the outer function; e.g. use "ab" rather than "gh" in the above program.

File, Display, and Keyboard Functions

The expression

```
"com.cm"/file
```

evaluates to a string which is the contents of the file name com.cm. If the file does not exist the value is fail.

The function write can be used to write a string onto a file.

```
"Hello there.^M" / write "comment.cm"
```

stores the string on the file. It is not possible to write on a file that has been read previously in the same Poplar session.

The function listout can be used to store any value on a file.

```
["a", "b"] / listout "f.pl"
```

stores ["a", "b"] on the file, including brackets and quote marks.

The expression

```
"f.pl"/listin
```

reads in the file f.pl and evaluates it as a Poplar expression. The file should have been created with the listout function.

The function print allows one to display a string.

```
"HI." / print
```

Displays HI. on the screen. The value is the input string.

The function key evaluates to a string which is the sequence of characters typed on the keyboard up to a RETURN. It prompts with its input string. Thus

```
"Type something, turkey!"/ key / print
```

echoes a line. Typing DEL to key will cause it to return fail.

The function dir produces a list of the file names currently in the system's directory. It ignores its input.

The function exec may be used to cause a command to be executed by the resident operating system command processor.

```
"ftp ivy st/c *.mesa" / exec
```

causes the command to be executed. Control should eventually return to the Poplar evaluator.

The Poplar Executive

The executive executes commands or evaluates expressions and prints their values. Input lines are

terminated with RETURN, M; if you wish to include a RETURN in the input line, precede it with a CTRL-V. For example,

```
"abc"M
```

is a valid (though trivial) program. Its value, "abc", will be printed. Common editing characters work during type-in. Backspace erases the previous character, and DEL cancels the input. Hitting ESC at the beginning of a line repeats the previous line typed; you can use backspace to change it. Hitting ESC followed by RETURN will repeat the previous command.

Most binary operators can omit their first operand and they will use the last value printed which we shall call the *current value*. For example

```
1M
```

prints the value "1". Then

```
+ 2M
```

prints "3".

Often, the previous value is wanted.

```
unM
```

throws away the current value and replaces it with the previous one. Only one value is saved. In the example above, the "1" would be displayed again. The effects of assignments or file writes are not undone.

The current value may be designated by @. For example,

```
Wind _ @M
```

will assign the displayed value to Wind.

After a certain number of lines of the current value are printed the message "... more ..." is typed at the bottom. If you want to see more type

```
moreM
```

and then type 'y' when subsequently prompted with "More?". If you type '&' to the More? question, it will print the entire string with no more pauses.

As you proceed, the cursor, which is in the shape of a square, will fill up to indicate how much memory space has been consumed. When it is mostly black, there will be a slight pause as the system "garbage collects" space not used any more. A smaller area on the left edge of the cursor indicates paging activity. A small mark moves each time the contents of a page buffer is changed.

If you've had enough type

```
quitM
```

Often, Poplar programs will be prepared using Bravo. The approved extension for such files is ".pl". Typing

```
$MyProgramM
```

will read in the filename and run it. It is shorthand for

```
"MyProgram.pl"/file/runM
```

The function run takes a string, analyzes it and evaluates it as a Poplar expression.

```
"1+2"/run = 3
"x _ ^"a^"; x x"/run = "aa"
```

and x assumes the value "a".

Often one says

```
MP _ "MyProgram.pl"/file; MP/runM
```

because it is useful to keep the program around as a string so that it can be editted using Poplar. This is slightly less painful than going back to Bravo, and a lot more fun. The most frequent kind of thing one types is

```
MP/{...("bad, old code" > "good, new code")...}M
```

or you may use subst:

```
MP/substM
```

which will prompt you for the new code and the old code (terminate by RETURN, as usual).

When you forget the above options, type

```
?M
```

to print out the user commands.

During type-in, the single ASCII quote (') can be used to enter strings:

```
'name/file means "name"/file
```

The string is terminated by any character not a letter, digit, period, or ^.

A running program may be interrupted by typing CTRL-DEL. The expression currently under evaluation is printed and you are talking to teh *debugging executive* which prompts with a $'. You are then given an opportunity to examine things further by the following mechanism: the debugging executive will evaluate any expression typed in the context of the evaluation. Thus local variables may be examined just by typing their names. If you just type a RETURN the interrupt message is repeated and a slightly larger expression context is displayed. If you type prM the computation resumes right where it left off. If you type DEL, the current computation is aborted and you return to the normal executive.

Input characters may be delivered to Poplar via the command line. Poplar starts by reading the command line (starting with the first blank) as if it were being typed in. When the command line is exhausted it begins to take its input from the keyboard.

If a command file contains

```
poplar "Running ex1"/print'
$ex1'
"Running ex2"/print'
$ex2'
quit
```

The effect of executing it will be to run the programs ex1 and ex2 and exit Poplar.

Errors

There are three kinds of errors - syntax errors, run-time errors, and unforeseen errors (Poplar bugs). A syntax error will give the line and character on that line near (but always after) where the error occurred, and will print the line involved. Lines are counted by counting carriage returns. The statement count is the number of semicolons passed in the entire program being compiled.

Run-time errors occur during the execution of programs and will print out the smallest expression being evaluated. You may then interact with the evaluator in the manner described above in the

paragraph on interrupts. However, typing prM will restart the computaion with the expression most recently displayed, rather than precisely where the error occured. This gives you the opportunity to fix things up a little and continue. Because it is not possible undo all the effects of a error this facility will not always yield the expected results.

A Poplar bug, is a "can’t happen" error message which indicates some internal inconsistency. It is treated just like a run-time error. Such errors should be reported to the Poplar implementors.

Poplar maintains various fixed-size storage areas which may overflow. Errors from storage overflow can be programmed around by breaking the program or its input into smaller pieces.

Odds and Ends

Poplar will create some temporary files beginning with "Poplar ..." and ending with "$". They may be removed and they will reappear when Poplar runs.

All transactions are recorded on the "Mesa.TypeScript" file.

Appendix A: Examples

The following examples are biased towards the kind of tasks Mesa programmers find themselves doing. This is more a function of the examples that come to our minds naturally, rather than the intent of the language.

Example 1. A pattern to eliminate Bravo format trailers

```
P _ {(...("^Z" ...)* "^M")! ...}
```

P says: Find all occurrences of "^Z" ... "^M" and delete the "^Z" ... part.

To transform old.bravo into new.text one types

```
"old.bravo"/file/P/write"new.text"
```

Example 2. Find all the mesa files for which a bcd file does not exist.

```
""/dir//tolower
//{... (".bcd" | ".mesa")}
/RelevantFiles := ["a.bcd", "foo.bcd", "ajax.mesa", "foo.mesa", "ed.mesa", "al.bcd",
                                "zug.bcd", "al.mesa"];
RelevantFiles//{... ".mesa"*} = ["ajax", "foo", "ed", "al"]
/MesaFiles:
RelevantFiles//{... ".bcd"*} /// ([x, y]: {y | x}) = {"zug" | {"al" | {"foo" | "a"}}}
/BcdPattern:
MesaFiles//{~BcdPattern ...} = ["ajax", "ed"]
```

The tolower maplist is required since Poplar considers "bcd" and "Bcd" distinct.

Example 3. Finding substrings in files.

```
FindAndShow _  (Pat: f: f/file/lines//{... Pat .../t: f ": " t/print; fail});
```

FindAndShow is a function that, given a string Pat, produces another function which takes a file name f and displays the lines f which contain Pat. Each line is prefixed with the file name. The final value is the empty list; the fail at the end is used to assure that printed lines are not saved.

Here is a program which searches mesa files a.mesa, b.mesa, and c.mesa.

```
"pattern:"/key/p ="PUBLIC":
p/FindAndShow     = (f: f/file/lines//{..."PUBLIC".../t: f ": " t/print; fail})
/FindPat:
["a", "b", "c"] ".mesa" // FindPat
```

Here is a program that prompts for the file names as well.

```
"pattern:"/key/FindAndShow/FindPat:
"" % (x: "file:"/key/FN: FN >FN/FindPat)
```

The prompt "file:" appears on the screen, and key returns the filename, FN. If the user types DEL, FN will be fail and everything stops.

Example 4. Print a set of files ending in ".pl" using Bravo, but save paper by combining them into one file and inserting headers.

```
Files _ ""/dir // tolower // {... ".pl"} / sort;
~(Files/isnull) >
        (Text _ Files // file "^Z^M^L^M";
        (("File:  " Files "^M^M^M") Text) / conc / (x: x "^Z^M") / write "out.out$";
        "Hardcopy out.out$" / exec;
        "out.out$" / delete)
```

Files is a list of the files to be printed. If Files is not the null list, it continues the computation. Text is a list of the contents of each of the files, appended with some Bravo formatting information to print each file on a new page. The third expression generates a list of headers for the beginning of each file, with carriage returns between the header and the file. That list is concatenated into a string, a final bit of Bravo formatting is added, and it is written on a dummy file out.out$. The Hardcopy command is executed, and then the file out.out$ is deleted.

Example 5. Automate the programming cycle.

```
Files _ ""/dir // tolower // {... ".errlog"*};
~(Files/isnull) >
        "Bravo/m " Files "; "/conc " del " (Files ".errlog "/conc)
        "; compile " (Files " "/conc) / quit
```

Files is a list of Mesa filenames for which a .errlog file exists. If Files is not the null list, each Mesa file and its .errlog are brought in with the Bravo m' macro, each of the .errlogs is deleted, and all those files are recompiled.

Example 6. The function takes a list as input, prints each list element and a question mark afterwards. If the letter y' is typed, that element will be an element in the resulting list, otherwise it will not.

```
Confirm _ (x: x//e: e "?"/key/{"y"}> e )
```

Example 7. Programs to add carriage returns to a paragraph

Assume that the input, Paragraph, contains spaces but no carriage returns. To make the example typographically tractable we shall limit lines to 20 characters. All spaces and carriage returns are inside strings are written explicitly. The algorithm goes as follows:

Initialize In to be Paragraph, and Out to be the empty string. As long as In is non-empty, Juggle In and Out so as to put another line on Out. Finally, return Out..

```
AddCrs1 _ (Paragraph := "0123^ 5678901^ 34^ 6789^ 1234567^ 9012^ 4567^ 9"
                        "01^ 3456789^ 123^ 56^ 89^ 1^ 345^ 789^ 12345^ 789"
                        "01^ 345^ 7^ 901^ 3456^ 8901^ 34^ 678^ 0123^ 56789"
                        "01^ 345^ 7";
        [Paragraph, ""]
        % ([In, Out]: In/{# ...} > [In, Out]/Juggle)
        /[In, Out]: Out

                        = "0123^ 5678901^ 34^ ^M"
                        "6789^ 1234567^ 9012^ ^M"
                        "4567^ 901^ 3456789^ ^M"
                        "123^ 56^ 89^ 1^ 345^ ^M"
                        "789^ 12345^ 78901^ ^M"
                        "345^ 7^ 901^ 3456^ ^M"
                        "8901^ 34^ 678^ 0123^ ^M"
                        "5678901^ 345^ 7"
                );
```

Juggle chops 19 characters off In, producing Line and RestOfIn. Then it breaks Line into everything up through the final blank, Most, and the remainder, Stub. The new value of In becomes the Stub followed by RestOfIn. The new Out becomes, Out followed by Most followed by a carriage return. If there are not 80 characters, In is set to be the empty string and Out to Out followed by In.

```
Juggle _ ([In, Out ]: := ["0123^ 5678901^ 34^ 6789^ 1234567^ 9.....",  "\\\"]

                In/
                        {[len 19, ...]  = ["0123^ 5678901^ 34^ 678",
                                                "9^ 1234567^ 9....."]
                        /[Line, RestOfIn]:
                        Line/{[(... " ")!, ...]} = ["0123^ 5678901^ 34^ ", "678"];
                        /[Most, Stub]:
                        [Stub RestOfIn = "6789^ 1234567^ 9.....",
                         Out Most "^M"= "\\\0123^ 5678901^ 34^ ^M"]
                        }
                | ["", Out In]
                );
```

The following recursive program is an alternative.

```
AddCrs2 _
        (Paragraph:
                Paragraph/
                        {[len 80, ...]
                                /[Line, RestOfLine]:
                                        Line/{[(... " ")!, ...]}/[Most, Stub]:
                                        Most "^M" (Stub RestOfLine/AddCrs2)
                        }
                | Paragraph
        )
```

Example 8. A program print all the files on the JuniperX directory which were written after the 9th of August. We assume that the file ftp.log has already been created as shown below. (The FTP subsystem will not execute a list command from the command line.)

```
Month _ {"Jan" > 01 | "Feb" > 02 | "Mar" > 03 | "Apr" > 04 | "May" > 05 | "Jun" > 06
          | "Jul" >07 | "Aug" > 08 | "Sep" > 09 | "Oct" > 10 | "Nov" > 11 | "Dec" > 12};

Date _ {[integer  "-"* ,  Month  "-"*  , integer]
       / [d, m, y] : y m (d/length/{2} > d | 0 d)};

File _ {(..."<JuniperX>")*  (word ">")!? word  ".mesa" ("!" number)*};

Line _ {[File   " "!* , Date   (...   "^M")*]};

Later _  ([f, d] : "9-Aug-78"/Date - d/{"-"...}>f);

"ftp.log"/file /f  :=     "....
                              <JuniperX>defs>BTreeDefs.mesa!3   8-Aug-78 18:05:13
                              <JuniperX>defs>FileSystemDefs.mesa!3   8-Aug-78 18:05:07
                              <JuniperX>defs>FileSystemDefs.mesa!4  8-Aug-78 18:05:15
                              <JuniperX>defs>triconprivatedefs.mesa!3  11-Aug-78 11:22:49
                              <JuniperX>hes>nelsonenv.mesa!3   4-Aug-78 13:59:26
                              <JuniperX>progs>CommonPineCold.mesa!3  11-Aug-78 17:36:24
                              <JuniperX>progs>eventmanager.mesa!2  11-Aug-78 17:41:41
                              <JuniperX>progs>eventmanager.mesa!3  11-Aug-78 11:40:39
                              <JuniperX>progs>eventmanager.mesa!4  11-Aug-78 11:44:17
                              <JuniperX>progs>wdisk.mesa!3 11-Aug-78 18:01:41
                               ....";
f / {Line,! ...*}  =     [  ["defs>BTreeDefs.mesa", 780808],
                              ["defs>FileSystemDefs.mesa", 780808],
                              ["defs>FileSystemDefs.mesa", 780808],
                              ["defs>triconprivatedefs.mesa",  780811],
                              ["hes>nelsonenv.mesa", 780804],
                              ["progs>CommonPineCold.mesa", 780811],
                              ["progs>eventmanager.mesa", 780811],
                              ["progs>eventmanager.mesa", 780811],
                              ["progs>eventmanager.mesa", 780811],
                              ["progs>wdisk.mesa",  780811]
                              ]
```

```
// Later =                 [  "defs>triconprivatedefs.mesa",
                               "progs>CommonPineCold.mesa",
                               "progs>eventmanager.mesa",
                               "progs>eventmanager.mesa",
                               "progs>eventmanager.mesa",
                               "progs>wdisk.mesa"
                               ]
/usort  =               [     "defs>triconprivatedefs.mesa",
                               "progs>CommonPineCold.mesa",
                               "progs>eventmanager.mesa",
                               "progs>wdisk.mesa"
                               ]
/ fileList :
"ftp ivy di/c juniperx ret/c "   (fileList  " " / conc)   "^M"
(fileList // { (word  ">")!?* ... }/ fileList:
   "bravo/h "   fileList   "^M" / conc
                ) =
                   "ftp ivy di/c juniperx ret/c  progs>CommonPineCold.mesa
                   progs>eventmanager.mesa defs>triconprivatedefs.mesa progs>wdisk.mesa
                   bravo/h CommonPineCold.mesa
                   bravo/h eventmanager.mesa
                   bravo/h triconprivatedefs.mesa
                   bravo/h wdisk.mesa
                   "

 /exec
```

Date produces numerical dates like 780809 for "9-Aug-78". The file is read in and broken up into a list of file-date pairs. All the things after the given date are filtered out and sorted, eliminating duplicates. This produces fileList. Then a giant Alto command is created which fetches the files and prints them using a Bravo macro.

Example 9. A cross reference program

The following program produces a cross reference listing for the files a.mesa, b.mesa, and c.mesa. It assumes that all the imported references begin with the prefix "P.".

```
Xref _ (FileList:
        FileList
          //(FileName := "a.mesa";
             FileName/file  =“..A1:PUBLIC ...P.B1...P.C2....A2: PUBLIC .. P.B1.....P.B4..."
             /{(((... "P.")* thing),! | []) ...*} = ["B1", "C2", "B1", "B4"]
             /usort = ["B1", "B4", "C2"]
             //(x: [x, FileName]) = [["B1", "a.mesa"], ["B4", "a.mesa"], ["C2", "a.mesa"]]
             /Imports:
             FileName/file
             /{((... / LastWord "PUBLIC"*),! | []) ...*} = ["A1", "A2"]
             //(x: [x, FileName "*"])
             /Exports:
                    Exports,, Imports = [["A1", "a.mesa*], ["A2", "a.mesa"*], ["B1", "a.mesa"],
                                                ["B4", "a.mesa"], ["C2", "a.mesa"]])
          ///([x,y]: x,,y) = [["A1", "a.mesa*"],
                                        ["A2", "a.mesa*"],
                                        ["B1", "a.mesa"],
                                        ["B4", "a.mesa"],
                                        ["C2", "a.mesa"],
                                        ["B1", "b.mesa*"],
                                        ["A2", "c.mesa"]]
          /factor =[["A1", ["a.mesa*"]],
                                ["A2", ["c.mesa"], ["a.mesa*"]],
                                ["B1", ["b.mesa*"], ["a.mesa"]],
                                ["B4", ["a.mesa"]],
                                ["C2", ["a.mesa"]]]
          //(x: x/1 " " (x/-1//1 " "/conc))
          /(x: x "^M"/conc) =
                                        "A1 a.mesa*
                                        A2 c.mesa a.mesa*
                                        B1 b.mesa* a.mesa
                                        B4 a.mesa
                                        C2 a.mesa
                                        ");
```

```
LastWord _ (s ="...A2 : ": s/reverse/{Sp* ":"* Sp* thing ...*}/reverse = "A2");

Sp _ {" "!?};

["a.mesa", "b.mesa", "c.mesa"]/Xref
```

Appendix B: Primitive Functions

The following primitive functions can be used in Poplar programs. A function is said to take as *input* the value it is applied to. It *returns* a value. A few functions take a *parameter*, which follows its name.

append filename

Appends its string argument to the file.

asort

Like sort (see below) but uses the ASCII collating sequence for all strings, including strings of numbers.

cfile

Functionally exactly like file, but copies the file into virtual memory. This allows one subsequently to overwrite the original file.

chop

Takes a string, breaks the string into single characters, and returns a list of strings of one character each.

```
"abc"/chop = ["a","b","c"]
```

check

Has the same effect as run, but checks the program as described in the section on equality assertions.

conc

Takes a list of strings as input and returns a string which is the concatenation of the list elements from first to last. A string as input will be returned as is. Thus, x///conc is equivalent to x/conc.

```
["a","b","c"]/conc = "abc"
```

daytime

Returns the day and time in a string.

```
""/daytime = "December 1, 1978  9:12 AM"
```

delete

Takes a string which is a file name on the local disk and deletes that file.

```
""/dir//{... "$"}//delete
```

deletes all files whose names end in "$", equivalent to "delete *$"/exec.

differ

Takes a list of two strings and compares them character by character. It returns a list of the two strings with any common prefix removed.

```
["abcX","abcy"]/differ = ["X","y"]
```

dir

Returns a list of file file names in the local directory. Ignores its input.

display

Prints its input on the screen. The input may be any Poplar value, e.g. a string, list, pattern. Returns its argument. See also print.

divide

Takes a list of two numbers [a,b] as input. Returns [a/b, a mod b]. If b is 0 it returns [a,a].

edit

Takes a filename as input, quits Poplar and executes Bravo, then after the user quits from Bravo, invokes Poplar executing the editted file.

exec

Takes any command, saves the current environment (via a Mesa checkpoint), has the Alto Operating System execute the command, and re-invokes Poplar in its saved state. exec returns its input.

factor

Takes a list of lists, sorts it according the the method of sort and then merges all adjacent sub-lists which have the same first element.

```
[[1,2,3], [3,4,6], [1,7], [3,8], [1,6,9], [9,0]]/factor =
[[1,[2,3],[7],[6,9]], [3,[4,6],[8]], [9,[0]]]
```

This function turns out to be quite useful for adding structure to data. See Example 9.

file

Its input is a string which is the name of a file. It returns a string with the contents of that file for future processing. See also listin.

ident

Does nothing and simply returns its input (the identity function).

islist

Returns its input if its input is a list, returns fail otherwise.

isnull

Returns its input if its input is the null list [], returns fail otherwise.

isstring

Returns its input if its input is a string, returns fail otherwise.

key

Prompts the typist with its input string. Waits for him to type in a sequence of characters terminated by RETURN. Returns the string without the RETURN. The Mesa ReadEditedString routine is used so the usual control characters can be used. If DEL is typed it returns fail.

length

Its input must be a string or a list. If its input is a string, returns the length of the string. If its input is a list, returns the number of list elements.

lines

Breaks the incoming string into a list of strings, one for each "line" or sequence of characters separated by carriage return. It is equivalent to

```
{((... "^M"),! | []) ,, [...]}
```

listin

Its argument is the name of a file created by listout. Returns the parameter to listout when the file was created, or fail if the file can't be processed.

```
listin = (x: x/file/x:x>x/run)
```

listout filename

Takes as input a Poplar expression, such as a list or string, and writes it in a special form on file filename. The expression may be recovered using listin.

marry

Takes a list of two lists of equal length. Each element of one list is paired with its corresponding element in the other list, and marry returns that list. See also zip.

```
[["a","b",1],["c","d",2]]/marry = [["a","c"],["b","d"],[1,2]]
```

max

Returns the maximum element of its input, which must be a list of numbers. Thus x///max is equivalent to x/max.

micas

A pattern which matches a single character and returns the number of micas it would take in Times Roman 10 pt. font.

```
"a"/{micas} = 165
"aA"/{micas,!} = [165, 265]
```

One can use scaling factors to get approximate answers for other fonts.

min

Like max, but the minimum.

minus

Takes a list of two numbers [a,b] as input. Returns a-b.

plus

Takes a list of two numbers [a,b] as input. Returns a+b.

print

Is like display but prints only strings. It prints them with no quotes. See display.

quit

Takes a string as input, executes the string as a command. Does not return to Poplar. See also exec.

reverse

Takes a list or string and reverses the order of its elements or characters.

```
["a","b","c"]/reverse = ["c","b","a"]
"abc"/reverse = "cba"
```

run

Takes a string, treats it like a Poplar program, and evaluates it, returning the value.

stop

Exits poplar like quit, but does not attempt to execute a command.

subst

Takes as input a string and performs substitutions for occurrences of a string (the pattern string). The *pattern* and the *substitution string* ("new" string) are prompted from the terminal. Typing DEL for either will abort the substitution. This function is similar in function to the Bravo Substitute command. It returns the input string with all occurrences of the pattern replaced by the substitution string. Typing

```
/subst M P M Q M
```

is equivalent to typing

```
/{(...("Q">"P"))!...} M
```

sort

Takes a list and returns its sorted permutation. If all the items are numbers the result is ordered by numerical value.

See also usort and asort.

symbol

Returns a list of all variables referenced or defined (sans primitives). Ignores its input.

times

Takes a list of two numbers [a,b] as input. Returns a*b.

tolower

Takes a string as input. Returns the string with all upper case letters [A-Z] changed to their lower case equivalents [a-z].

toupper

Takes a string as input. Returns the string with all lower case letters [a-z] changed to their upper case equivalents [A-Z].

usort

Uses asort to sort the incoming list, which must be a list of strings, and returns a sorted list with duplicate strings removed. See also asort, sort.

write filename

Takes the incoming string and writes it on file filename. If file filename exists, it will be overwritten. An error will occur if the file is open for reading. The simplest way to avoid such errors is to never write on a file which as previously been the input to a file or listin. See also listout.

zip

Takes a list of two lists. Elements in the two lists are interleaved in the resulting list.

```
[[1,2],[3,4]]/zip = [1,3,2,4]
```

The following functions are used in debugging Poplar and are not normally useful or necessary.

garbage

Forces a garbage collection. Ignores its input.

snap

Gives a snapshot of the storage allocator. Ignores its input.

Appendix C: Syntax Equations

This grammar describes how the Poplar parser treats a program. Therefore it describes many illegal programs whose illegality does not come to light until the program runs. Don't be confused by the names given to grammatical categories. They are meant to be suggestive of the first one or two alternatives, but not all of them. Thus a *Function* can, degenerately, be a *Sequence* which can be a *Statement*, etc.

Non-terminals are in *italic*, literal characters are in **bold**

PoplarProgram **::=** *Function* **end-of-file**

Function ::= *Variable* **:** *Function*
| **[** *VariableList* **] :** *Function*
| *Prelude Function*
| *Sequence*

Prelude ::= *Statement* **;**
| *Variable* _
| *ChoiceExp* **|**
| *ThenExp* **>**
| *BinaryExp BinOp*

Sequence ::= *Statement* **;** *Sequence*
| *Statement* **;**
| *Statement*

Statement ::= Variable _ *Statement*
| *ChoiceExp*

ChoiceExp ::= *ChoiceExp* **|** *ThenExp*
| *ThenExp*

ThenExp ::= *ThenExp* **>** *BinaryExp*
| *BinaryExp*

BinaryExp ::= *BinaryExp BinOp PrefixExp*
| *BinaryExp PrefixExp*
| *PrefixExp*

BinOp ::= ***/*** | ***//*** | ***///*** | **%** | **+** | **-** | **,,** | **--**

PrefixExp ::= *UnaryFunction PreFixExp*
| **-** *PrefixExp*
| **~** *PrefixExp*
| *PostFixExp*

PostFixExp ::= *PostFixExp* **!**
| *PostFixExp* **,!**
| *PostFixExp* **?**
| *PostFixExp* *****
| *SimpleExp*

SimpleExp ::= *String*
| *Variable*
| *PrimitiveFunction*
| **@**
| **#**
| **fail**
| **...**
| **[** *List* **]**
| **[ ]**
| **{** *Function* **}**
| **(** *Function* **)**

List ::= *List* **,** *Function*
| *Function*

VariableList ::= *VariableList* **,** *Variable*
| *Variable*

Variable ::= a single letter | any sequence of letters and digits including one capital letter

UnaryFunction ::= **len** | **blanks** | **write** | **listout**

String ::= " any sequence of characters in which a ^ precedes every " "

PrimitiveFunction ::= a sequence of two or more small letters (see Appendix B.)

Poplar uses these literal symbols:

; / // /// % | > : + - ,, -- ~ ! ,! ? * " @ # ... [ ] { } ( ) ,

Appendix D

Getting Started

Get [maxc]<morris>poplar.image and run it.

On [maxc]<morris>pl> is a set of files ex1.pl - ex9.pl for each of the examples in Appendix A.

Please send comments to Morris.