Paul McJones
5 Oct 78

DRAFT: October 4, 1978 3:35 PM

We are hoping to release this in a week or so. Please read this and tell us how to make it clearer. Some of the things written are not quite true yet, so if you want to try using Poplar let me know. -Jim

Poplar

Jim Morris
Eric Schmidt

The Poplar language is intended for text manipulation. It is based on ideas from LISP, SNOBOL, and APL. It is experimental in the sense that we do not know quite how to use it. Certain of your favorite features have been left out or given strange implementations in order to encourage exploration of different programming styles. This manual describes the current implementation. Read this, try it, and give us some feedback. Example programs, successful or not, will be greatly appreciated. Comments, especially about capabilities we do *not* offer, are solicited.

This manual is composed of several sections:

It has some appendices:

We recommend new users read the main part once over and then follow the directions in Appendix D.

Strings and Lists

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

```
"I am a string"
"I am a string with a
carriage return in it"
""
```

are three different strings. The last 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 to represent characters which are inconvenient to type into strings:

```
↑" 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 carriage 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 surrounded by brackets and separated by commas. For example,

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

are lists, the last being the empty list, or *nil.* 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.

The major difference between strings and lists is that lists have a structure that makes it faster to pick out the pieces. It will be faster to extract "625 B St." from the list structure above than to find the same information in the string

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

For the list you simply say "Get the second item in the third sublist". For the string you would have to say something like "Go to the third bracket. Go to the first comma. Skip the space. Get all the characters up to the next comma."

Function Application

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

There are many built-in 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 operation 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 applying the list to a number, as if the list were a function. The numbering starts with 1.

Compare with Backus's selector functions

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

Applying a list to a negative number results in the list *minus* that number of elements.

```
-1/["ab", "c", "d"] = ["c", "d"]
-2/["ab", "c", "d"] = ["d"]
-3/["ab", "c", "d"] = []
```

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 or numerical order.

```
[4, 1, -2]/sort = [-2, 1, 4]
["beta", "zero", "alpha"]/sort = ["alpha", "beta", "zero"]
```

If the element of a 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", 100], ["jane", 3]]/sort = [["al", 100], ["fred", 20], ["jane", 3]]
```

The function usort, ("unique sort"), sorts a list of strings and then eliminates duplicates.

```
["x", "a", "x", "b"]/usort = ["a", "b", "x"]
```

There is a complete list of all Poplar built-in functions (called *routines*) in Appendix B.

Conditional Branching

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

```
[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, also can 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 differing values for an expression in a variety of ways. The relevant operators for doing this are |, and >.

The *alternation* operator, |, has the following behavior.

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

Thus

```
(x/isstring) | "def"
```

will be x if x is a string, but will be "def" otherwise.

The operator > is a *conditional value* operator with the following behavior

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

Thus

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

will be "xyz" if x is "abc", but will be "def" otherwise.

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 operations 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
```

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

```
["a", "bcd", ""]//length = [1, 3, 0]
[]//length = []
```

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

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

The operation // may be used in conjunction with -- to produce sublists.

```
3--6 // [10, 20, 30, 40, 50, 60, 70] = [30, 40, 50, 60]
```

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.
```

which 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 = 34
[2, 4, 8]///minus = -10
["The ", "quick ", "brown ", "fox "]///conc = "The quick brown fox "
```

More generally,

```
[x1, x2, x3, x4] /// f = [[[x1, x2]/f , x3]/f, x4]/f
[x]///f = x
[]///f = []
```

Variables, Assignment, and Sequencing

A *variable* is either a single letter or a sequence of letters including 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"
```

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

```
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. When this occurs it is often desirable to combine them into sequences and ignore the values they produce. The semi-colon is used to separate such items.

```
(x ← 1; y ← 3; x+y) = 4
```

Forms

A *form* is a variable or list of variables followed by a colon followed by any expression.

```
x: x+1
[x, y, z]: (x/length) + (y/length)  z
```

Forms may be applied to values or lists of values and the effect is to substitute the values for the corresponding names.

```
["a", "bc", "ttt"] / ([x, y, z] :  (z/length) + (y/length)  x x)
```

yields

```
("ttt"/length) + ("bc"/length)  "a" "a"
```

which eventually yields

```
"32aa"
```

The operators `//` and `///` may be used as well

```
["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 form may be assigned to a variable and then that variable may be referenced like a built-in function.

Patterns and Matching

Patterns are used to analyze strings. In general, a pattern is 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"/{"ab"} = fail
```

Patterns may be combined with the operator "|" (*alternation*) to form an new pattern which matches either the first or second component.

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

Patterns may be juxtaposed to form a new pattern which matches strings formed by concatenation of strings which 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 sucessfully. 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 ~ (*not*) changes fail into the empty string and anything else into fail.

```
"abc" / {~"abc"} = 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 = {"." integer | (integer ("." integer?)?)}
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 = {" " | "↑l"}
```

A few other patterns can be modified by a number

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

The number is called the pattern's *parameter.*

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 forms. 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.

Note:
X ≡ (X:X/isfail/isfail)

```
"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"]
```

Like any other operator the application operator may appear 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"
```

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

```
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)}

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

The idiom {...(P > S)...} means replace the first occurence of P with S.

The idiom {(...(P > S))!...} means replace all occurences 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 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 greater or fewer characters so that P2 *would* match the remainder. Nevertheless, the matcher does not bother trying new ways to match P1.

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"!. Thus the elipsis 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 elipsis can be surprisingly expensive occaisionally. 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
```

You can write recursive functions, if you like.

```
Blanks ← (n: (n/{0}) > "" | ("↑ " ((n-1)/Blanks)))
```

which generates a string of n blanks.

You can also write recursive patterns

```
AE ← {number | ( "(" AE ("+" | "-") AE ")" ) }
```

should match something like "(5-(6+2))".

One can write a fairly succint evaluator for arithmetic expressions

```
Eval ← {number |
            ([ "("* Eval "+"*,  Eval ")"* ]/plus) |
```

```
([ "("* Eval "-"* , Eval ")"* ]/minus)};
```

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

File, Display, and Keyboard Operations

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 operation write can be used to write a string onto a file.

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

stores the string on the file. The value is the input string.

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 operation.

The operation listout can be used to store any Poplar value (not just a string) on a file.

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

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

The operation display allows one to display a string.

```
"Hl." / display
```

Displays "Hl." on the screen. The value is the input string.

The name key evaluates to a string which is the sequence of characters typed on the keyboard up to an ESC. Thus

```
key / display
```

echoes lines. Typing DEL to key will return fail.

The name dir evaluates to be a list of the file names currently in the system's directory.

The special name 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 might eventually return to the Poplar evaluator. The value of the expression is the input.

There is a complete list of built-in operations in Poplar in Appendix B.

Precedence and Scope

A program is one or more *statements*, separated by semicolons. A statement is either an assignment statement, whose right hand side is an *expression*, or simply an expression. Expressions come in three types: binary, unary, and simple. A simple expression is, for example, a string, a list or a parenthesized expression. Unary (one-argument) expressions are often used in patterns, for example, "a"! is a unary expression composed of the single expression "a" and the operator "!". Binary (two-argument) expressions look like 1+2, or "a" > "b".

With all the different operators, it might be hard to determine which operation comes first in a complicated expression. Poplar has the following rules:

Parentheses, brackets and braces control the order of evaluation regardless of the precedence of operators.

It is never a mistake and often a good idea to add extra parentheses.

Unary operations are performed sooner than binary operations.

```
"a"|"b"*  =  "a"|("b"*)
```

There are two levels of precedence for binary opertors.

The operators /, //, ///, and % are performed after all the other binary operations. They are viewed as "control" operations, not "expression" operators.

Between binary operators of the same precedence, the leftmost is executed first.

```
S/{"ab"|"xY"} | "a"  =  S / ({"ab"|"xY"} | "a")

1-2-3  =  (1-2)-3
"a"  >  "b" | "c"  >  "d"  =  (("a" > "b") | "c")  >  "d"

x/y//z  =  (x/y)//z
```

The symbol ':' encompasses everything that follows.

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

```
x: t ← x/conc; x+1      =      x: (t ← x/conc; x+1)

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

is the same as

```
Succesor ← (x: x+1; Predecessor ← (x: x-1));
```

Thus one will usually enclose forms by parentheses.

The Poplar Executive

The executive has two display windows. It executes commands typed into the lower window, and prints the results in the upper window. As in Bravo, ESC, *not carriage return,* terminates all input. For example,

```
"abc"
```

followed by hitting the ESC key is a valid (though trivial) program. Its value, "abc", will print in the upper window. 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.

Common editing characters work during type-in. Backspace erases the previous character, and DEL cancels the input. Hitting ESC followed by ESC will execute the previous command.

To the executive, most binary operators can omit their first operand and they will use the value displayed in their upper window. For example

```
1
```

followed by ESC, prints the value "1" in the upper window. Then

```
+ 2
```

followed by ESC, prints "3" in the upper window.

Often, the previous value is wanted:

```
un
```

followed by ESC throws away the value in the upper window and replaces it with the one that was there before. Only one value is saved. In the example above, the "1" would be displayed again.

The value displayed in the upper window may be designated by @. For example,

```
Wind ← @
```

will assign the displayed value to Wind.

When the value in the upper window won't fit the message "... more ..." is typed at the bottom. If you want to see more type

```
more
```

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.

Quite possibly, you will find yourself in the Mesa debugger. If you type q, carriage return, you should get back to Poplar. Please notify us of the "PBug" error message.

If you type

```
reset
```

everything is reinitialized, as if you had just entered Poplar.

If you've had enough type

```
quit
```

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

```
$MyProgram
```

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

```
"MyProgram.pl"/file/run
```

The built-in 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/run
```

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")...}
```

or you may use the built-in function subst:

```
MP/subst
```

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

When you forget the above options, type

```
?
```

followed by ESC to print out the user commands.

N.B. 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 ↑.

Errors

There are three kinds of errors - syntax errors, runtime errors, and unforeseen errors (PBugs). 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.

Runtime errors occur during the execution of programs and will print out the smallest expression being evaluated. The question "Bigger context?" will be asked. Typing 'y' will print the containing expression.

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

The third type of error, a PBug, is a "can't happen" error message which indicates some internal inconsistency. The Mesa Debugger will be invoked (if it is on your disk). The best thing to do is record the message, quit out of the debugger, and report the error to us.

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.

Because scrolling does not work in the rather primitive Poplar windows, strings may or may not expand and print in full, depending on the situation. Strings which are abbreviated on the screen have seven dots (.....) between the beginning and ending substrings. Routines like print and uniq can be used to print the entire string.

All transactions in the lower window 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.

Example 1. A program to eliminate Bravo format trailers

```
P ← {(...("↑Z" ...)* "↑M")! ...}
```

P says: Find all occurences 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.

```
RelevantFiles ← dir//tolower//{... (".bcd" | ".mesa")};
```

Produces something like

```
["a.bcd", "foo.bcd", "ajax.mesa", "foo.mesa", "ed.mesa", "al.bcd", "zug.bcd", "al.mesa"]
```

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

```
MesaFiles ← RelevantFiles//{... ".mesa"*};
```

produces

```
["ajax", "foo", "ed", "al"]
```

```
BcdPattern ← RelevantFiles//{... ".bcd"*} /// ([x, y]: {y | x});
```

produces something like

```
{"a" | {"foo" | {"al" | "zug"}}}
```

Finally

```
MesaFiles//{~BcdPattern ...} = ["ajax", "ed"]
```

Example 3. Finding substrings in files.

```
Find ← (Pat: f: [f],, (f/file/lines//{... Pat ...}))
```

Find produces a list of all the lines in file f which contains Pat. The list starts with the file name.

Here is a program which searches all the mesa files on a directory

```
("pattern:"/print/key)/Find/FindPat:
dir // tolower // {... ".mesa"} //FindPat
```

The first line prompts the typist and creates a function FindPat which will search for it. For example if he typed "FooBar", FindPat would be

```
(f: [f],, (f/file/lines//{... "FooBar" ...}))
```

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

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

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 // (x:x "↑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. Get a list of all the files on your disk, sorted by their size.

```
dirlength // ([x,y]:[y,x]) / sort / reverse
```

dirlength returns a list of lists [name, size in bytes]. Switch those two and sort them in ascending order. Reverse that list. This is a very slow program.

Example 7. Programs to add carriage returns to a paragraph

Assume that the input, Paragraph, contains spaces but no carriage returns. 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:
                    [Paragraph, ""]
                    % ([In, Out]: (In/{# ...}) > [In, Out]/Juggle)
                    /([In, Out]: Out));
```

Juggle chops 80 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]:
                    (In/
                            {[len 80, ...] /[Line, RestOfIn]:
                            Line/{[(... " ")!, ...]}/[Most, Stub]:
                            [Stub RestOfIn, Out Most "↑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.

```
"ftp/l ivy di/c JuniperX li/c '*.mesa,c↑Mti↑M↑Mq"/exec;
```

This procduces a file, ftp.log that looks something like

```
....
<JuniperX>defs>BTreeDefs.mesa!3   8-Aug-78 18:05:13
<JuniperX>defs>FilePageUseDefs.mesa!3   8-Aug-78 18:05:07
<JuniperX>defs>FileSystemDefs.mesa!3   8-Aug-78 18:05:15
<JuniperX>defs>JuniperFileDefs.mesa!3   8-Aug-78 18:05:18
<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>CommonPineHot.mesa!3  11-Aug-78 17:39:58
<JuniperX>progs>ControlHelper.mesa!3  11-Aug-78 17:41:41
<JuniperX>progs>eventmanager.mesa!3  11-Aug-78 11:40:39
<JuniperX>progs>HTABLES.mesa!3  11-Aug-78 11:44:17
<JuniperX>progs>Intentions.mesa!3  11-Aug-78 16:34:13
<JuniperX>progs>machinesync.mesa!3  11-Aug-78 16:25:44
<JuniperX>progs>Vectors.mesa!3  11-Aug-78 16:38:14
<JuniperX>progs>wdisk.mesa!3 11-Aug-78 18:01:41
....
```

The rest of the program works on this file.

```
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)))};
```

Date produces numerical dates like 780809 for "9-Aug-78"

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

Line ← {[File " "!* , Date (... "↑M")*]};
```

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

"ftp.log"/file
        / {Line,! ...*}
        // Later
        /usort
        / fileList :

        "ftp ivy di/c juniperx ret/c "   (fileList  " " / conc)   "↑M"

        (fileList // { (word  " > ")!?* word }/ fileList:
          "bravo/h "   fileList   "↑M" / conc
        )
    /exec
```

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, elinating duplicates. This produces fileList. Then a giant Alto command is created which fetches the files, 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, c.mesa and d.mesa. It assumes that all the imported references begin with the prefix "P.".

```
Xref ← (FileList:
        FileList
         //(FileName:
                FileName/file
                /{((... "P.")* word),! | [] ...*}
                /usort
                //(x: [x, FileName])
                /Imports:
                FileName/file
                /{((... / LastWord) "PUBLIC"*),! | [] ...*}
                //(x: [x, FileName "*"])
                /Exports:
                Exports,, Imports)
          ///([x,y]: x,,y)
          /List:
                  [[[""]]],,(List/sort)   /// Merge / reverse /(x:-1/x)
                  //(x: x " ")
                  /(x: x "↑M"/conc);
LastWord ← (s: s/reverse/{Sp* ":"* word ...*}/reverse);
Merge ← ([MasterList, Item]: 1/(1/MasterList)/cItem:
                    ((1/Item)/{cItem})
                            > ([Item,,(-1/(1/MasterList))],,(-1/MasterList))
                           |  ([Item],, MasterList)
              );
Sp ← {" "!?};
["a.mesa", "b.mesa", "c.mesa", "d.mesa"]/Xref
```

For every file on the list FileList we first produce the list Imports that looks something like

```
[["C2", "a.mesa"], ["B1", "a.mesa"], ["B4", "a.mesa"]]
```

for the file "a.mesa". Then we produce the list of exported names which might look like

```
[["A1", "a.mesa*"], ["A2", "a.mesa"]]
```

All of these lists (two for each file) are then concatenated. This master list is then sorted by

procedure name, and adjacent lists with the same name are merged. For example, the master list

```
[["A1",  "a.mesa*"],
["A2",  "a.mesa*"],
["A2",  "c.mesa"],
["C2",  "a.mesa"],
["B1",  "a.mesa"],
["B1",  "b.mesa*"],
["B4",  "a.mesa"]]
```

would Merge to

```
[["A1",  "a.mesa*"],
["A2",  "c.mesa",  "a.mesa*"],
["C2",  "a.mesa"],
["B1",  "b.mesa*",  "a.mesa"],
["B4",  "a.mesa"]]
```

Finally, the list structure is converted to a single string with carraige returns and spaces.

Appendix B: Index to Routines

The following built-in functions, called *routines*, can be used in Poplar programs. A routine is said to take as *input* the value it is applied to. It *returns* a vale. Some routines take a *parameter*, which follows the routine's name.

The general form is

```
    input/routine = result
or  input/routine parameter = result,
```

where / may also be //, ///, or %.

Some routines ignore their input, and some return a trivial result, such as their input. If a routine ignores its input it can begin a line, as if it were a value. For example, one can type

```
dir/sort
```

rather than something silly like

```
""/dir/sort
```

Finally, some routines take only strings as input. The maplist (//) or gobble (///) operators allow these routines to be applied over each element of an input list.

asort

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

chop

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

Example:

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

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.

Example:

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

confirm

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, if 'n' is typed it will not.

debravo

Takes as input a string and returns a string with sequences of ↑Z followed by anything up to but not including carriage returns removed. This removes Bravo formatting information. It is equivalent to

```
{(...("↑Z" ...)* "↑M")! ...}
```

delete

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

Example:

```
"scavengerlog"/delete
```

also

```
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.

Example

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

dir

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

dirlength

Like dir returns a list, but each list element is a list of two elements, a filename and its length in characters. Files whose length is undeterminable have length 0. Ignores its input.

As each file must be opened to get the file's length, this command is very time consuming.

display

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

divide

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

edit

Takes a filename as input, quits Poplar annd 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 reinvokes Poplar in its saved state. exec returns its input.

> The saved checkpoint may not work so return to Poplar is uncertain. To be certain state should be explicitely saved with listout.
>
> Since there is no way in Mesa to tell if this is a restart or not, the exec routine will ask the question "Quit?" in the lower window. You should type 'n', for no. Then when the system restarts itself you should respond to the same question 'y', for yes.

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 write, listin, and listout.

ident

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

isfail

Returns the empty string if its input is fail, returns fail otherwise.

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

Waits for the user to type in a sequence of characters terminated by ESC or DEL. Returns that string. This can be used to interact with a Poplar program. No explicit prompt is printed, but, e.g. "prompt:"/print/key will explicitely prompt. Ignores its input.

last

Returns the last element of its input list, or its input if it is not a list.

```
L/last = L/length/L
```

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 identical to

```
S/{(... "↑M"),! | [] ,, [...]}
```

for string S.

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.

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. This gives a primitive but reliable checkpoint facility.

loop

Takes a string, considers it a Poplar program, and runs the program indefinitely. See also run, step.

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. Example:

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

max

Returns the maximum element of its input, which must be a list of numbers.

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 is guaranteed only to print strings and prints them with no elipses. See also uniq, display.

quit

With no input simply quits Poplar. With 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.
Example:

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

run

Takes a string, thinks of it as a Poplar program, and executes it once. See also loop, step.

step

Takes a string, thinks of it as a Poplar program, and executes the program in a "single step" mode, in which step prints out each statement and waits for confirmation before executing it. See also loop, run.

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 routine is similar in function to the Bravo Substitute command. It returns the input string with all occurences of the pattern replaced by the substitution string.

sort

Takes a list and returns a sorted list. If the list is a list of strings, they are sorted according to the sorting rules below. If it is a list of lists, the first element of the sublist must be a string and the lists will be sorted with those strings as keys. The comparison between strings is as follows:

If the elements being compared are strings with digits 0-9 only, they are assumend to the numbers and are sorted according to the integer value of the number.

If the elements are not all digits, they are compared using the ASCII collating sequence.

See also usort and asort.

symbol

Returns a list of all variables referenced or defined (sans built-ins). 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].

uniq

Like display except all special characters and control characters are explicit. The input must be a string. See also print and display.

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. Returns the input string as its result. See also file, listin, and listout.

zip

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

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

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

close

Closes all open files. Normally unnecessary since reset and quit close all files. Ignores its input.

garbage

Forces a garbage collection. Ignores its input.

snap

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

Appendix C: Syntax Equations

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

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

Program ::= *Statement* ; *Program*
| *Statement* ;
| *Statement*

Statement ::= **variable** ← *Expression* | *Expression*

Expression ::= *Expression* / *Factor*
| *Expression* // *Factor*
| *Expression* /// *Factor*
| *Expression* % *Factor*
| *Factor*

Factor ::= *Factor* | *Term*
| *Factor* > *Term*
| *Factor* : *Program*
| *Factor* + *Term*
| *Factor* - *Term*
| *Factor* ,, *Term*
| *Factor* -- *Term*
| *Factor* *Term*
| *Term*

Term ::= **routine** *Term*
| - *Term*
| ~ *Term*
| *Term* !
| *Term* ,!
| *Term* ?
| *Term* *
| *Primary*

Primary ::= "string"
| variable
| routine

```
        | @
        | #
        | fail
        | ...
        | [ List ]
        | [ ]
        | { Program }
        | ( Program )

List ::= List , Program
        | Program
```

Poplar uses these graphic tokens:

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

But not =

Appendix D

Getting Started

Get [ivy]<morris>poplar.image (or poplar.bcd which can be fed to basicmesa) and run it (you need a Mesa 4.1 disk).

Begin by typing in a few strings and lists to get a feel for the system. Try reading in a file:

```
"user.cm"/file
```

Then break it into a list of lines using

```
/lines
```

This list is easily sorted

```
/sort
```

and finally made into a string again

```
/conc
```

Typing

```
un
```

will make it back into a list.

In this way, the novice Poplar user can try a number of different Poplar commands and interactively program an algorithm. Once the commands are clear, the file Mesa.Typescript can be edited and the relevant commands put int a .pl file for execution.

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

Please send comments to Morris.

Hints to the Novice

1. Don't forget to type ESC after commands to the Poplar executive. *Carriage return* will not work.

2. Upper- and lower-case letters are ALWAYS DIFFERENT to Poplar. toupper and tolower are useful in this regard.

3. The workings of the alternation operator '|' in patterns {} are murky- best is to have longest items first.

4. Don't forget the '{' and '}' around patterns.

5. Always put '(' and ')' around user-defined functions (:).

patch:
exec: Exec + (c:'Rem.cm/(f:f/file(s:""/close;c 'RunMesa4.0Poplan.CheckPoint.Image$/exec;
0. quit question is a pain! (s/write f;c))\
?use "intention" file?
1. should close files before checkpoint
a) 'rem.cm/file;c/exec' gets PBug b) "f/file;"copy f+e"/exec;'f/file gives wrong result
2. $ suffix on .image requires explicit RunMesa.run prefix
3. should restore original Rem.cm contents at end

- Need comment convention
- need better editing facilities
- symbol:
  ?initial values all meaningfull : opt, seq o/c, seq o f
  ?spelling: plural
- $, more, // couldn't be written in Poplan; "eval quote"; client-at-a-time input; binary op syntax
- ? Poplan executive should display value of assignments
- Bottom window needs ..... mechanism for error printouts (?long lists)
- functional programming
  unified syntax: abstraction, tupling, unary/binary/etc. operator
  "apply all to" (like Backus's "u-functions/construction) (then get sublists by "apply to all m--n")
  composition, cond, while
- []/isnull should be [] (but is "")
- $[]_{0,0}[1,2]$ is disallowed ("applied to nonlist")
- ↑ should work for other graphics, for use in ' literals
- loop prints value each time around?
- loop, step: should apply (quoted) to expressions?
- [[],[1]]/[x,y]:(x/isnull)>y|x,y = [[1],[]]
1. Blinker cursor only when that window is accepting input.
2. // preserve fail; weed ← (L:L/isnull>[]|1/L/isfail>-1/L|[1/L],,-1/L/weed)
3. list patterns, e.g. weed ← {[(... fail x)!? ...]}
   [x,y]: is a form of list pattern
5. drop chop; add ,!? (Then chop={#,!?})

- Examples on [Ivy]<Morris>pl> inconsistent with Appendix A: 3,7 4,8 5,3? 6,4 7,5 8,6 ?
  + bugs