MSPM SECTION BZ.10.02

IDENTIFICATION

APL Formal Syntax and Reductions

PURPOSE

This document describes precisely the allowed syntax of input lines to the Multics APL interpreter when it is in the immediate execution mode. It also specifies the reduction rules which drive the parser.

This document does not describe the syntax of system requests (lines beginning with a right parenthesis), nor the syntax accepted in function definition mode (which differs slightly due to header lines, labels, and recognition of the nabla character). In fact, lines entered in function definition mode are not syntactically analyzed at the time; they are only lexically analyzed. The lexical analyzer outputs a token string which does obey the syntax rules stated here, and it is this string which is actually parsed at execution time.

TOKENS

Five kinds of tokens are output by the lexical analyzer. They are:

- BEGINNING-OF-LINE. This is supplied by the lexical analyzer prior to all other tokens.
- END-OF-LINE. Similar to BOL, this token follows all other tokens.
- CONSTANT. All constants, whether alphabetic or numeric, scalar or vector. The token carries with it means of access to the type and value of the constant.
- NAME. All variable names, function names, and stop/trace control names. The token carries the literal string which is the name.
- OPERATOR. All other constituents of the source line are considered single character operators.

Constants are typed and converted to internal format by the lex, though neither this nor the rank of the constant is of syntactic concern.

Names are not typed at lex time, but are left as literal strings. At parse time, the referent of a name will be determined when it is first encountered, and this will type it as a variable, a dyadic function (two arguments), a monadic function (one argument), a zero-adic function (no arguments), or a stop/trace control. Whether a function returns a value or not is not of syntactic concern. Since the referent of a name may change unpredictably (due to editing during a suspension), when the reference is evaluated, the type of the name is again checked to ensure that it has not changed; if it has changed a syntax error is signalled.

Some operators are typed into classes by the lexical analyzer. There are five classes:

- SOP--scalar operator. Members are + - & ÷ "ce "fl * "lo ⌊ ! "ci.
- MOP--mixed operator. Members are ? % , $ "tr "rf.
- LOP--logical operator. Members are "an "or "na "no < ≤ = ≥ > ≠.
- DOP--dyadic operator. Members are "up "do "ep "ev "en and backslash-hyphen.
- GOP--grade operator. Members are "gu "gd.

All other single-character tokens remain unclassified; that is, each may be said to be a class by itself.

The lexical analyzer also discards the entire contents of lines which begin with the lamp symbol (comment indicator). Such lines come to the parser looking like [redacted] BOL EOL.

BASICS

In summary, the above considerations give rise to thirty basic symbols for the purposes of syntactic analysis. They are:

con	gop	[
var	bol	]
zfn	eol	;

mfn	≠	"qu
dfn	/	"qq
stc	"ib	.
sop	tilde	"cc
mop	"rr	{
lop	(	}
dop	)	backslash

CATEGORIES

The syntax of APL is very simple. It contains only five categories:

- COP--compress/expand operator, / and backlash.
- VAL--value.
- EXP--expression.
- LIST--expressions concatened with semicolons.
- S--statement. A complete line.

BNF

The actual syntax rules are given here. Alternatives are listed on successive lines.

```
cop:    /
        \

val:    con
        var
        zfn
        "qu
        "qq
        ( exp )
        val [ ]
        val [ list ]

s:      bol eol
        bol } eol
        bol } list eol
        bol list eol
```

```
list:   ;
        exp
        ; exp
        list ; exp
        list ;

exp:    "qu { exp
        var { exp
        var [ ] { exp
        var [ list ] { exp
        stc { exp
        val sop exp
        val lop exp
        val mop exp
        val dop exp
        val dfn exp
        val ⁄ exp
        val cop exp
        val "rr exp
        val sop . sop exp
        val sop . lop exp
        val lop . sop exp
        val lop . lop exp
        val "cc . sop exp
        val "cc . lop exp
        val cop [ exp ] exp
        val "rr [ exp ] exp
        val tilde exp
        mfn exp
        "ib exp
        sop exp
        mop exp
        gop exp
        "rr exp
        sop / exp
        lop / exp
        sop / [ exp ] exp
        lop / [ exp ] exp
```

(exp continued)

sop / exp
lop / exp
gop [exp] exp
"rr [exp] exp

The manner in which the syntax is expressed in these rules has no particular merit other than that it readily leads to the reductions analysis rules. In particular, the above syntax is not simple precedence.

REDUCTIONS ANALYSIS SCHEME

A reductions analyzer is an automated syntax parser, driven by a table of rules. Each rule has two parts: a pattern and an action. The analyzer has a rule location counter, indicating the rule currently being processed, and a stack, on which to push down basics and category tokens while building up syntactic units. The analyzer takes as its input the stream of tokens from the lexical analyzer, which it reads left-to-right with no backing up. The next token to be read, however, can be inspected without being read (or, if you like, it is read but held in a one-token buffer).

Initially, the analyzer starts on the first rule with its stack empty. Its operation is then:

- The pattern portion of the current rule is compared with the current stack contents and token to be read next. The rule pattern contains a (possibly null) list of stack entries and a (possibly null) basic. The pattern is said to match if stack list matches the top of the stack as far as the list goes (a null list matches any stack), and the basic matches the token next to be read (if the basic in the rule is null, the next to be read is not inspected).
- If the pattern match fails, the rule location counter is merely advanced to the next rule.
- If the pattern match succeeds, then all of the following happen:
  - If the next token to be read was matched to a basic in this rule, the token is read (i.e., is not accessible any more).

- The action specified in the action portion of the rule is done. This action can be one of four things. (1) ERROR: a syntax error is recognized. Processing of the current line is aborted. (2) DONE: successful recognition of a complete and correct statement has occurred. (3) PULL: the basic just matched, hence, read, is pushed onto the stack. (4) promotion to a category: the group of stack entries matched by the pattern of this rule are popped from the stack and replaced by a single entry of the given category name.
- The rule location counter is set to the rule designated as the one handling the current stack top. Each possible stack top has associated with it a (fixed) rule to be processed next.

The reduction rules of the APL interpreter obey the following restrictions:

- Actions of DONE or ERROR occur only on rules with null patterns (i.e., rules which always succeed), and rules with null patterns have only actions of DONE or ERROR.
- Actions of PULL occur only on rules with non-null basics, and rules with non-null basics have only actions of PULL.
- Actions of promotion to a category occur only on rules with null basics but non-null stack patterns, and conversely.

REDUCTION RULES

The following are the actual reduction rules used by the APL parser. The labels down the left indicate which rule handles each stack top (i.e., the rule transferred to upon creating a new stack top of the given kind). The dollar-sign symbolizes the top of the stack, with stack contents listed to the left, farther to the left the deeper in the stack. The symbol to the right of the dollar sign is a basic to match the next token to be read. The action column lists the action as either DONE, ERROR, PULL, or simply the new category name for promotion to a category. For example, the rule after the one labelled "sop" would be interpreted as follows: if the next token to be read is a period, and if the stack top is a SOP, and if the

next-to-the-top entry in the stack is a VAL, then pull the period into the stack (pushing down SOP, VAL, and everything below), and go to the rule labelled "period"; otherwise, go to the next rule.

One other point needs explanation. The abbreviations "Lexp" and "Llist" occurring in the basic pattern slot mean that any token which is ultimately permitted as the leftmost token of an expression or, respectively, a list, is to be accepted. The members of the "Lexp" set are:

con	(	lop
var	tilde	mop
zfn	mfn	gop
"qu	"ib	"rr
"qq	sop	stc

The members of the "Llist" set are the members of "Lexp" plus a semicolon.

```
start:      PULL                          $ bol
bol:        PULL                          $ eol
            PULL                          $ }
            PULL                          $ Llist
            ERROR
con:        val                     con $
zfn:        val                     zfn $
"qq:        val                     "qq $
mfn:
tilde:
"ib:
dfn:
dop:
mop:
/:
(:
{:          PULL                          $ Lexp
            ERROR
var:        PULL                          $ {
            PULL                          $ [
            val                     var $
```

```
stc:    PULL                          $$ {
        ERROR

sop:    PULL                          $$ Lexp
        PULL                  val sop $$ .
        PULL                          $$ /
        PULL                          $$ ≠
        ERROR

lop:    PULL                    val lop $$ Lexp
        PULL           val sop . lop $$ Lexp
        PULL                    val lop $$ .
        PULL           val "cc . lop $$ Lexp
        PULL                          $$ /
        PULL                          $$ ≠
        ERROR

gop:    PULL                          $$ [
        PULL                          $$ Lexp
        ERROR

eol:    s                   bol } eol $
        s                bol } list eol $
        s                  bol list eol $
        s                      bol eol $
        ERROR

s:      DONE

/:      PULL                  sop / $$ Lexp
        PULL                  lop / $$ Lexp
        PULL                  sop / $$ [
        PULL                  lop / $$ [
        cop                       / $$

\:      cop                       \ $$

"rr:    PULL                          $$ Lexp
        PULL                          $$ [
        ERROR

):      val                ( exp ) $

[:      PULL                  var [ $$ ]
        PULL                  var [ $$ Llist
        PULL                  val [ $$ ]
        PULL                  val [ $$ Llist
        PULL               val cop [ $$ Lexp
        PULL                   "rr [ $$ Lexp
        PULL                 sop / [ $$ Lexp
        PULL                 lop / [ $$ Lexp
        PULL                   gop [ $$ Lexp
        ERROR
```

```
]:      PULL                    var [ ] ## {
        PULL                var [ list ] ## {
        val                     var [ ] ##
        val                 var [ list ] ##
        val                     val [ ] ##
        val                 val [ list ] ##
        PULL            val cop [ exp ] ## Lexp
        PULL                "rr [ exp ] ## Lexp
        PULL              sop / [ exp ] ## Lexp
        PULL              lop / [ exp ] ## Lexp
        PULL                gop [ exp ] ## Lexp
        ERROR

;:      PULL                            ## Lexp
        list                    list ; ##
        list                         ; ##

"qu:    PULL                            ## {
        val                        "qu ##

.:      PULL                            ## sop
        PULL                            ## lop
        ERROR

"cc:    PULL                            ## .
        ERROR

}:      PULL                            ## eol
        PULL                            ## Llist
        ERROR

cop:    PULL                    val cop ## Lexp
        PULL                    val cop ## [
        ERROR

val:    PULL                            ## [
        PULL                            ## sop
        PULL                            ## lop
        PULL                            ## mop
        PULL                            ## dop
        PULL                            ## dfn
        PULL                            ## /
        PULL                            ## /
        PULL                            ## \
        PULL                            ## "rr
        PULL                            ## "cc
        exp                        val ##

list:   PULL                            ## ]
        PULL                            ## ;
        PULL                            ## eol
        ERROR

exp:    PULL                      ( exp ## )
        PULL                val cop [ exp ## ]
        PULL                val "rr [ exp ## ]
        PULL                  sop / [ exp ## ]
        PULL                  lop / [ exp ## ]
        PULL                    gop [ exp ## ]
        PULL                    "rr [ exp ## ]
```

(exp continued)

```
list              list ; exp $$
list                   ; exp $$
exp                 "qu { exp $$
exp                 var { exp $$
exp           var [ ] { exp $$
exp       var [ list ] { exp $$
exp                 stc { exp $$
exp             val sop exp $$
exp             val lop exp $$
exp             val mop exp $$
exp             val dop exp $$
exp             val dfn exp $$
exp               val / exp $$
exp             val cop exp $$
exp             val "rr exp $$
exp        val sop . sop exp $$
exp        val sop . lop exp $$
exp        val lop . sop exp $$
exp        val lop . lop exp $$
exp        val "cc . sop exp $$
exp        val "cc . lop exp $$
exp      val cop [ exp ] exp $$
exp      val "rr [ exp ] exp $$
exp                tilde exp $$
exp                  mfn exp $$
exp                  "ib exp $$
exp                  sop exp $$
exp                  mop exp $$
exp                  gop exp $$
exp                  "rr exp $$
exp                sop / exp $$
exp                lop / exp $$
exp        sop / [ exp ] exp $$
exp        lop / [ exp ] exp $$
exp                sop / exp $$
exp                lop / exp $$
exp          gop [ exp ] exp $$
exp          "rr [ exp ] exp $$
list                     exp $$
```

It is notable that the above reduction rules will reduce "var [ ]" and "var [list]" into "val", while there is no corresponding BNF rule. This is done to avoid backtracking in the parse. The reduction rules delay promotion of VARs to VALs until it is certain that they are not left sides of assignments.

TYPE CODES

This is a complete list of the 36-bit type codes used by the lexical analyzer and the parser. The codes are given in octal.

400000000000	beginning-of-line
200000000000	end-of-line
100000000000	constant
040000000000	stop/trace control
020000000000	name (note 1)
020000000000	variable (note 2)
010000000000	zero-adic function (note 2)
004000000000	monadic function (note 2)
002000000000	dyadic function (note 2)
001000000000	scalar operator
000400000000	mixed operator
000200000000	logical operator
000100000000	dyadic operator
000040000000	grade operator
000020000000	backslash-hyphen
000010000000	/
000004000000	backslash
000002000000	"ib
000001000000	tilde
000000400000	"rr
000000200000	(
000000100000	)
000000040000	[
000000020000	]
000000010000	;
000000004000	"qu
000000002000	"qq
000000001000	period
000000000400	"cc
000000000200	left arrow
000000000100	right arrow
000000000040	other (note 3)
000000000020	compress/expand operator (note 4)

000000000010	value (note 4)
000000000004	expression (note 4)
000000000002	list (note 4)
	statement (note 5)

Note 1: Since names are not typed by the lexical analyzer, all name tokens are given type code 020000000000.

Note 2: When a name token is PULLed into the parse stack, the actual kind of name will be determined, and one of these four type codes will be assigned.

Note 3: Any other single character appearing in the input stream is assigned this type. This token will ultimately cause a syntax error in the parse.

Note 4: These four type codes represent categories; hence, they are not output by the lex, but only internally generated in the parse by promotion actions.

Note 5: Since end-of-statement is detected in the actions for the EOL rules and the parse stopped there, promotion to type "statement" actually never occurs, and no type code is assigned.

Note that with the above type code assignments, the masks for "Lexp" and "Llist" become:

175643606000	Lexp
175643616000	Llist