MSPM SECTION BZ.10.03

Identification

APL Lexical Analyzer

Purpose

The lexical analyzer accepts as input the canonicalized source line typed to the APL intrepreter, breaks it up into the syntactically meaningful tokens (identifiers, constants, etc.), and classifies the tokens. Constants are converted to internal format.

Entry Points

The lexical analyzer has two different entry points: one for use in immediate execution mode to lex user input to S- and E-frames, and the other to lex function lines as they are stored away at the close of a function definition. Statement labels are recognized and tabulated by definition mode lex; they are not recognized by execution mode lex.

Operation

In operation, the lexical analyzer is called by either the parser or the editor when a new line is needed, after any needed go-ahead characters have been output. The caller supplies a pointer to the top of the state indicator stack, upon which the lex will build the resultant token list. If the current call is to lex an S- or E-input line, the tokens will remain there until the frame is discarded. If the current call is from a function definition, the token list will be moved to the function storage segment when the line has been completely processed.

The format of each token on the token list is:

```
1 t  based          token
   2 p  offset      next token pointer
   2 m  bit(36)     token type mask
   2 s  fixed       source char index
   2 t  fixed       constant type
   2 n  fixed       constant number
   2 rr  fixed      constant rhorho
```

```
                    2 r(1:t.rr) fixed   constant rho
                    2 v  cell           constant  value
                       3 b  bit(t.n)    bit constant
                       3 c  char(t.n)   char constant
                       3 i(1:t.n) fixed int constant
                       3 f(1:t.n) float bin(63)   float constant
```

Here, t.m is the token type code as specified in Type Codes, MSPM BZ.10.02; t.s is the index within the source line of the first character which comprised this token (for error reporting); t.t through t.v.f are the value of the constant if t.m is constant; t.v.c is the actual source characters if t.m is name, stop/trace control, or any of the various operators.

The detailed action sequence is:

- A beginning-of-line token is placed on the stack.
- If this is a function mode call, the input line will be provided by the caller. If this is an execution mode call, the lex will read a source line from the current input stream. The line will be placed in t.v.c of the beginning-of-line token.
- If the source line begins with nabla or right parenthesis, immediate exit is made to the editor or the request processor.
- If the source line contains an unmatched quote not preceded by an unquoted lamp-symbol (comment indicator) further source lines will be read and appended to the string in t.v.c until the quotes pair.
- The character index is initialized to zero. It indexes characters of the source line.
- BLOOP (BUMP, then LOOP): BUMP is performed. BUMP consists of adding one to the character index, extracting the character at that position of the source line, and classifying the character.
- LOOP: The location of the current token is saved, and the token pointer is advanced over the current token to where the next can begin.
- Control is dispatched to the handler of the classification of the current source character.

Name and Operator Handling

The following is the handling of alphabetic and underscore characters:

- The CLEAN operation is performed. CLEAN is used to clean-up the format of a preceding constant token; see the discussion of CLEAN and constant handling later on.
- The present source character index is remembered; the delta-switch is reset.
- A loop is entered which consists of:
  - BUMP.
  - For alphabetics, numerics, and underscores, continue loop.
  - For delta, verify that the delta is the second character of the token and that the first was "S" or "T". If so, set the delta-switch and continue. If not, consider the name ended and exit from the loop.
  - For all other characters, exit from the loop.
- Create a new token, of type name unless the delta-switch is set, in which case of type stc (stop/trace control).
- Place in t.v.c of the token the source characters between the saved source character index and the current source character index.
- Go to LOOP, which, by not BUMPing, will reprocess the character which terminated the name.

The following is the handling of blanks and tabulates:

- Go immediately to BLOOP, to fetch and process the next character.

The following is the handling of new-line and lamp characters:

- The CLEAN operation is performed.
- An end-of-line token is produced.
- Return is made to the caller of the lex.

Handling of scalar operators, mixed operators, logical operators, grade operators, and dyadic operators is essentially alike:

- The CLEAN operation is performed.
- A token is created, of type sop, mop, lop, gop, or dop.
- Go to BLOOP.

Handling of break characters which are types by themselves consists of:

- CLEAN.
- A token of appropriate type is created.
- Go to BLOOP.

Constant Handling

Now we come to the handling of constants. APL allows a vector of constants to be input, simply by concatenating successive values with blanks. Such a vector must appear as one *constant* token in the token list. There are two problems associated with this. First of all, it is unknown what the data-type of the final vector will be (it may start with zeros and ones, suggesting type *bit*; but be followed by small integers, necessitating promotion to type *int*; and then finally a non-integral value can force promotion to type *float*). Second, since isolated values are to be considered scalars and not vectors of length one, the rhorho of the constant is also unknown in advance. Character constants share the second problem with numeric constants, though not the first.

These two problems are overcome as follows: the token which represents a constant is not finalized until the lex detects that the constant has finally ended. In the meantime, the constant is stored as if it were to be a vector, and, if numeric, it is stored as type *float*. Additional element values keep appending to the same token. Finally, when some token other than a constant is to be placed in the token list, subroutine CLEAN will be called to properly format the constant token.

CLEAN performs as follows:

- Exit if the previous token is not a constant.
- Exit if the previous token is a character constant having other than one element.
- If the previous token is a character constant of exactly one element, it is made a scalar (rhorho = 0 instead of 1), and then exit.
- Now the token is known to be a numeric constant. If the final type can be *bit* or *int*, loop through the element values converting them from *float*. This can be done in place because those types are smaller in size than *float*. The actual conversion is a simple substr operation.

- Finally, if the constant consists of exactly one element, rhorho is set to zero.

Now the handling of constant-class characters can be discussed in detail.

The handling for quote is:

- If the preceding token is not of type <u>constant</u>, fabricate a <u>constant</u> token, of data-type <u>char</u>, having no characters in it.
- If the preceding token is a constant, but not character, then CLEAN first, and then fabricate a character token as above.
- Otherwise, the preceding token is also a character constant, so append to it.
- L:Remember the present character index plus one.
- Advance the character index to the next quote.
- If the character following the current character is not also a quote, append the characters from the saved character index up to but not including the present character to the character token and exit to BLOOP.
- If the next character is also a quote, then append the characters including the current character (one quote), advance the character index by one to skip the next quote, and go back to L.

The handling of the upper-minus sign is as follows:

- The present source character index is remembered; S, the numeric sign switch, is set to minus; V, the 20-digit numeric value accumulator, is cleared; P, the digits-after-point counter, is cleared; E, the exponent accumulator, is cleared; isw, the ignoring-digits switch, is turned off; and psw, the point-seen switch, is turned off.
- DLOOP: The main digit-loop is entered.
- BUMP.
- Dispatch on the new character.
    - If the new character is a period, test the point-switch. If on, go to ENDNUM. If off, turn it on and return to DLOOP.
    - If the new character is "E", go to the exponent processor.
    - If the new character is anything but a digit, to to ENDNUM.

- Now the new character is known to be a digit. Test the ignore-switch to see if it should be ignored (digits are ignored after 20 significant digits have been accumulated). If so, subtract one from P. Then test the point-switch; if on, add one to P. Then return to DLOOP.
- If the ignore-switch is off, then verify that V, the 20-digit accumulator, is less than 10**20. If so, V=10*V+digit; test the point-switch and add one to P if on.
- If the accumulator is not less than 10**20, then set the ignore-switch. If the current digit is 5 or more, add one to V. Subtract one from P to record the ignored digit; and, if the point-switch is set, add one to P.
- Return to DLOOP.

If a digit is encountered, the handling is:

- The present source character index is remembered; S is set to plus; V, P, and E are set to zero; and the point-switch and ignore-switch are turned off.
- The digit-loop is entered at the V=V*10+digit operation.

If a period is encountered, the handling is:

- The next character of the source line is inspected to see if it is a digit.
- If not, this period is not part of a number. A token is created for it by itself.
- If so, the present source character index is remembered; S is set to plus; V, P and E are cleared; the ignore-switch is turned off and the point-switch is turned on.
- Digit loop is entered at DLOOP.

The exponent processor works as follows:

- ES, the exponent sign, is set to plus.
- BUMP.
- Anything other than a digit or a sign is a SYNTAX ERROR at the current character.
- A minus sign sets ES to minus.

- ELOOP: Either sign comes here. A BUMP is performed. Anything but a digit causes E=ES*E and exit to ENDNUM.
- A digit, either from the previous step or from the first BUMP, causes E = 10*E+digit.
- E is tested against 1000; if greater, DOMAIN ERROR.
- Return is made to ELOOP.

At ENDNUM, the number is converted to normalized double floating-point:

- E = E-P. The effective exponent is the expressed exponent plus the number of ignored digits minus the number of digits following the point.
- The value accumulator is decimal-normalized so that 10**20< V $\leq$ 10**21. E is adjusted accordingly.
- Unless E<19 at this point, DOMAIN ERROR.
- Unless -61< E at this point, the number is taken as zero.
- The 20-digit accumulator is fractionally multiplied by a fixed-point Eth power of ten, 71-bits precision, normalized to have a one in bit 1 (example: when converting the digit one, the value accumulator will contain 10**21, E will be -21, the table entry -21 will contain (10**-21)*(2**69)).
- E is converted to the binary exponent: E = 71-normalization of above table entry (in the case of the digit one, E = 71-69 = 2).
- The value accumulator is binary normalized. If V<2 **70, then V = 2*V and E = E-1.
- The value accumulator is rounded to precision 63: if the 2**7 bit is set in V then if V< 2**71-2**8 then V = V+2**8, otherwise V = 2**70 and E = E+1.
- The sign of the number is applied: if S is minus, then if V< 2**70+2**8 then V = -V+(2**8-1), otherwise V = -2**71 and E = E-1.
- Unless -129< E at this point, the number is taken as zero.
- Unless E< 128 at this point, DOMAIN ERROR.
- The result is 8-bits of E concatenated with the 2**70...2**8 bits of V.
- If the result is 0 or 1, note that the value can be of type bit.

- If $-1 \leq E < 36$ and all bits (E+9)...71 are zero, note that the value can be of type int.
- Otherwise, note that the value must be of type float.
- Unless the previous token is a constant token, create a constant token, of data-type as noted above, store the value (as float regardless of the type determined). Go to LOOP.
- If the previous token is a character constant, then CLEAN it up and create a new constant token as in the previous step.
- Otherwise, the previous token must be a numeric constant. Append the current value to the values in it.
- Promote the type of the constant if the present type is longer than the type stored with the constant token.
- Go to LOOP.

All Other Characters

All other characters found in the source line becomes tokens of type other. Ultimately, they will be rejected by the parser.