MSPM SECTION BZ.10.05

Identification

APL Data Formats

Purpose

This section describes the internal layout of data used by the Multics APL interpreter.

Introduction

The data base of the APL interpreter is the currently active workspace. The date of the current workspace is distributed among four segments in the user's process directory. The segments are named "apl.stack.?", "apl.symbol.?", "apl.value.?", and "apl.function.?", where "?" represents an identifier unique to each invocation of APL.

The Stack Segment

The stack segment contains the APL state indicator. Its format has already been discussed in the parser and lexical analyzer descriptions. It consists of a succession of S-, E-, or F-frames. S- and E-frames contain also a token list; all contain a parse stack.

The Value Segment

The value segment is managed as a PL/1 area with value nodes explicitly allocated and freed by ALLOCATE and FREE statements. The value area contains values and lists of values. The format of a value bead is:

```
1 value based
  2 count fixed
  2 type fixed
  2 number fixed
  2 rhorho fixed
  2 rho(1:value.rhorho) fixed
  2 data(1:number) ? unaligned
```

where "?" is "bit(1)" if value.type is 1=<u>bit</u>, "fixed" if value.type is 2=<u>integer</u>, "float bin(63)" if value.type is 3=<u>float</u>, and "char(1)" if value.type is 4=<u>character</u>. Value.number is the actual number of elements of this data value (multiplication-reduction of rho of the value, in APL terms). Value.rhorho is the rank of the value. Value.rho is the rho of the value, always stored in 1-origin indexing.

Value.count deserves more extended explanation. It is a count of the number of pointers in the entire workspace which point to this value. A multiplicity of such pointers may exist, because, for reasons of efficiency, the interpreter often supplies duplicate pointers to one value rather than copying values over. The problem then arises as to when to free the storage occupied by the value itself. This problem is solved by value.count. Each time a new copy of a pointer to a value is created, the count in that value is increased by one. As each user of a value is finished with it, he subtracts one from its count. The last user will notice that the count has gone to zero and will deallocate the value.

In general, wherever the phrase "deallocate the value" occurs in the documentation of the APL interpreter, it must be understood to mean: subtract one from the usage count, and actually free the storage only if the count has reached zero.

A list is represented by a thread of structures of the form:

```
1 list based
   2 next offset
   2 value offset
```

where list.next is the offset (relative to the value area) of the next element of the list, and list.value is the offset of the value constituting this element of the list. Lists have no usage counts,because they can have only one pointer to them active. Lists occur only as subscripts and mixed-mode output statements.

The Symbol Segment

The symbol segment mechanizes the symbol table. Symbols are hashed into 87 buckets, which are maintained as linearly threaded lists from an array of 87 starting pointers. Beyond the starting array, the symbol segment consists of an area in which are allocated spelling beads, usage beads, and group beads. Spelling beads are strung from the buckets; each identifier of a given spelling has exactly one spelling bead. Usage beads are strung from each spelling bead to reflect the fact that a spelling can have more than one usage.

The format of a spelling bead is:

```
1  spell based
   2 next offset
   2 usagep offset
   2 spelling char(256) varying
```

where spell.next continues the hash bucket thread; spell.usagep is the offset of the head of the thread of usage beads hanging from this spelling bead; and spell.spelling is the actual spelling itself.

The format of a usage bead is:

```
1  usage based
   2 count fixed
   2 type fixed
   2 valuep offset
   2 spellp offset
```

where usage.count is the number of pointers that refer to this bead (like value.count, discussed above); usage.type is 0=junk, 1=variable, 2=function, or 3=group; usage.valuep is the offset of a value in the value area for a variable, a function bead in the function area if a function, or a group bead in the symbol area if a groupl and usage.spellp is the offset of the spelling bead owning this usage bead.

The format of a group bead is:

```
1 group based
  2 names char(infinity) varying
```

where the names of the members of the group are found in group.names separated by single blanks.

The meaning of usage.type=junk deserves comment. During the parsing of lines to be interpreted, names must be bound to the referents to which they refer! In particular, the referent of a name is some usage bead--the one which corresponds to the proper generation of the identifier in question. However, after a name is initially bound to a referent but before that reference is evaluated, the original referent can be changed or deleted by the APL user in definition mode (example: the identifier "A" is encountered in parsing an input line; the parser establishes its referent as a function and hooks it to the proper usage bead; the right argument of the function reference is yet to be evaluated; the right argument is a call on function "B", which suspends;

the user then erases "A" and resumes execution; the right argument now being available, the interpreter returns to execute "A"; whoops, no "A" anymore!).

Whenever an editing operation alters the meaning of a usage bead with a non-zero usage count, the bead is set to type=junk. A new usage bead of the proper new type is created, if necessary. When the holders of the pointers to the old bead come back to it and discover it has been junked, they will re-search the symbol table in an effort to establish another referent for the identifier in question. If the new referent is of compatible type with the old, the replacement succeeds and execution continues. If not (examples: a function changes into a group, or simply disappears), a syntax error is signalled.

The Function Segment

The function segment is an area in which are allocated function beads, line beads, and token beads.

There is one function bead for each defined function. The value pointer in the usage bead of the function's name (in the symbol area) points to the function bead. The format of the function bead is:

```
1 fb based
  2 nargs fixed
  2 nresults fixed
  2 nlines fixed
  2 slb                    source line bead
    3 next offset
    3 prev offset
    3 class fixed
  2 lines(1:fb.nlines)
    3 source offset
    3 lex offset
```

where fb.nargs is the number of arguments this function has (0,1, or 2); fb.nresults is the number of results (0 or 1); fb.nlines is the number of lines (NLs occuring within quotes are not counted in this number); fb.slb is the header of a doubly-threaded list of source line beads (see below); fb.lines is the line array, one entry per line, with fb.lines(i).source being the pointer to the first source line bead for line i, and fb.lines(i).lex being a pointer to the beginning-of-line token for line i.

The tokens for each line are singly-threaded off fb.lines(i).lex, and have the format described in the lexical analyzer description (MSPM BZ.10.03).

Source line beads are doubly-threaded from the fb.slb header. There is one source line bead for each NL in the source text of the function. The format of each source line bead is:

```
1 slb based
   2 next offset
   2 prev offset
   2 class fixed
   2 lineno fixed
   2 line char(infinity) varying
```

where slb.next is a pointer to the next source line bead (slb.next in the last source line points back to fb.slb); slb.prev is a pointer to the previous source line bead (slb.prev in the first source line points back to fb.slb); slb.class is an integer signifying 0=EOF, 1=CON, 2=END, 3=NEW, to be explained below; slb.lineno is the APL line number assigned to this line during editing scaled four decimal places; slb.line is the actual source line itself, ending in NL.

This slb.class variable is used to keep track of which lines need to be lexically analyzed again when a function has been edited. Class EOF occurs only in the fb.slb header bead to indicate the end of the source lines. Class NEW occurs only temporarily during editing. All the lines of a stored function are classed as either CON (continued) or END (not continued). The last source line corresponding to each lex line is classed END; all previous source lines belonging to that same lex line are classed CON (CON lines appear due to NLs which occur in character constants).

Usage of the class NEW is described in the documentation of the editor. See MSPM BZ.10.06.