LITTLE Newsletter 37 — Edith Deak

Proposal for MIDL (GLITTLE) — 9/74

This newsletter will describe a proposed major extension to LITTLE, which we provisionally designate as MIDL. The MIDL language is intended for

1. implementation of the SETL optimizer (for which LITTLE appears too limited)
2. the writing of SETL-compatible new primitives, when SETL programs are to be brought to 'production' levels of efficiency by an essentially 'manual' procedure.

The MIDL language will provide:

(a) pointers, a garbage-collected memory millieu, recursive calls; all compatible with the present SETL garbage collector.

(b) features facilitating communication with SETL.

As far as possible MIDL should preserve the machine independence which characterizes LITTLE and SETL.

Literature: Newsletter 73; SETL specification in
Installment II of On Programming;
Item 6 of Installment I of On Programming.

Detailed Language Specifications follow.

1. Data objects, heap blocks, pointers, hash tables.

Several basic new semantic objects will be added to LITTLE. These include

i. Pointers (to heap blocks)
ii. Code addresses (for supporting recursion).

Pointers will be discussed in this section; code addresses in Section 4.

Heap blocks will have the formats described in Item 6, p. 50 (of Installment I of On Programming). Our aims are the following:

(a) to avoid word-length dependencies (and, in particular, involvement with the question of the number of pointers that can be stored in a word)

(b) to avoid explicit restrictions on the position of pointers within a word (by leaving these positions flexible, we may hope to exploit field-related special operators available on one or another machine)

(c) to make it easy to communicate with the SETL run-time library

(d) to attain reasonably high efficiency (for this, a machine-oriented peephole optimizer may be required). This central requirement will keep us closer to the 'low level' semantic approach of LITTLE than would otherwise be suitable.

(e) to allow the programmer to deal in a conveninent way with objects possessing large numbers of miscellaneous attributes. For this to be accomplished successfully, our language will have to include mechanisms which avoid 'name conflicts' between the names of attributes of objects of different types.

MIDL should be upwardly compatible with LITTLE. All existing dictions in LITTLE, including the macro facility, will have the same semantics and syntax in MIDL.

Our specific approach is as follows:

i. We introduce several new primitive types of data object, extending the fundamental bit string of LITTLE. An object can be either *atomic*, a (1-dimensional) *array object, a maptable*, or a *SETL object*.

ii. Atomic objects are either bitstrings (of stated size); integers (signed and of implementation-determined size and internal format); real numbers (of implementation-determined size and internal format); pointers (of implementation-determined size); *structures* (see below); and (procedure) *entries*.

A pointer is of an implementation-determined size, and points to a heap object.

Variables are declared in the form

(1) DCL $name_1$ $typename_2$, ..., $name_k$ $typename_k$; ...;

Here, $name_j$ names a variable, and $typename_j$ gives its type.

Types are introduced by <u>type definitions</u>.
The predefined types are BITS, REAL, PTR, SETLOBJ, ENTRY.
New types are introduced by *type declarations*. The simplest form of type declaration, which introduces a structure, is

(2) TYPE typename: partname_1 typedesc_1,...,partname_n typedesc_n;

An example would be

```
(3)   TYPE LISTNODE: PREV PTR(LISTNODE),
                     NEXT PTR(LISTNODE), VALUE BITS(10);
```

In (2), *typename* is any admissible LITTLE name, which becomes the name of the type introduced by (2); $partname_1$,...,$partname_n$ are valid names, all distinct, which come to name the components of the type introduced by (2). The syntactic objects $typedesc_1$,...,$typedesc_n$ are all *type descriptors*.
A type descriptor will either be of one of the forms REAL, BITS(n), where n is an integer constant, ENTRY, or will be a *pointer type descriptor* having one of the forms

(4a) PTR(typedesc')

(4b) PTR

(4c) MAP (ns,typedesc)

(4d) SETLOBJ

The *typedesc'* which follows the word PTR in (4a) has a somewhat different structure than an ordinary *typedesc*. Specifically, we allow the construction

(5) PTR (* typedesc)

(which describes a pointer to an array of elements, all of type *typedesc*). A pointer declared in the form (5) is called an *array pointer*. A pointer declared by (4b) is called an *unqualified pointer*; a pointer declared by (4a) is called *qualified*.

To access a component of the object pointed to by a qualified pointer V, one writes

(6a) partname V

if V points to a non-array structure;

(6b) V(index)

if V points to an array, or

(6c) partname V(index)

To define the nature of the object pointed to by an unqualified pointer V *qualification operators* are provided. Qualification operators have the syntactic form

(7) t :: V ,

where t is either a type name or a type descriptor. An occurrence of V in the context (7) is understood to be an object of type t.

An object of type PTR $typedesc'_j$ can be assigned to any variable declared to have the type (4b); and conversely.

By using qualification operators, the fields of such an object can be retrieved subsequently and treated properly. As an example, suppose that P is a variable of type PTR and that R is a variable of type PTR(* tr) .
Then the assignments

(8a) P(j) = R(k) and (8b) R(k) = P(j)

are legal. Moroever, after the assignment (8a) the expressions

(9) *field* tr::P(j) and *field* R(k)

retrieve the same quantity (here, we assume that *field* designates some component name that has been declared for structures of type *tr*). Finally, we observe that the sequence of the assignments

```
(10)          P(j) = R(k);
              field tr::P(j) = X;
```

is legal, and has the same effect on P as

```
(11)          field R(k) = X;
              P(j) = R(k);
```

Next, suppose that P1 and P2 are declared as PTR(t), where t is defined by:

```
(12)          TYPE t: f1 BITS(16), f2 PTR(t);
```

That is, t is a structure which consists of a 16 bit field and a pointer to a structure of type t. The assignment

(13) $P1 = P2;$

will set P1 to point to the same heap object pointed to by P2.

The assignment

(14) $P1 = f_2 P_2$

sets P1 to point to the structure referenced by the field f2 in the structured referenced by P2.

As an additional convenience making it easy to transfer all the fields of one structure to the fields of a variable of identical structure, we introduce the diction

(15) $\uparrow V$

which accesses the whole of a non-array object pointed to by a pointer V. The form (15), like the forms (6a-c), may be used on the left-hand side of an assignment statement.
Thus, the assignment

(16) $\uparrow P1 = \uparrow P2;$

sets the value of the structure referenced by P1 to the value of the structure referenced by P2. Finally, the assignment

(17) $\uparrow(f_2\ P1) = \uparrow P2;$

sets the value of the structure pointed to by f2 of P1 to the value of the structure referenced by P2.

The option of explicitly dereferencing pointers to access components of structures is also provided by the forms:

(18a) partname ↑ V

(18b) ↑ V (index)

(18c) partname ↑ V(index) .

Semantically, dictions 18(a-c) are equivalent to 6(a-c) respectively.

An expression of the form

(19) F V

is legal if and only if V is of a declared or qualified type which has a component named F. Otherwise the compiler will issue a diagnostic.

Storage in the heap for structures must be explicitly allocated. A non-array heap object of type t is created by a function call of the form

(20) NEW(t)

An array heap object with n components of type t is created by a function call of the form

(21) NEW(t,n)

The function NEW returns pointers to the heap block allocated. The length n of an allocated object is obtained by use of the prefix operator .NELT.

All fields of the newly created block are initialized to zero, and all pointers to the system undefined atom, *omega*, which may be written in a source program as the symbol .OM. (see Section B on interface with SRTL).

To reduce the size of the heap array object referenced by pointer P, eliminating all but the first n of its components, we can use the function (note: with a side effect on P)

(22) TRIM(P,N)

Heap blocks as we have introduced them give a quite acceptable 'dynamic array' capability, and thus make it possible to deal comfortably with functions defined on a dense range of integers. However, we find ourselves in quite a different situation in attempting to deal with a function defined on a sparse range of integers. In SETL this is no problem, since the general 'mapping' concept which SETL provides handles sparsely defined functions in quite a reasonable way. The technique underlying this SETL primitive is of course hashing. The most customary lower level techniques for dealing with sparsely defined mappings are not necessarily superior to hashing. For example, the use of arrays or lists in which argument values x are coupled with functional values f(x) is common, but this can lead to quite inefficient implementations of value retrieval and modification. Faced with a sparsely defined map, a programmer attempting to achieve efficiency by working in a low level language will often attempt the invention of *ad hoc* encodings or data arrangements which expedite access to map-values. However, in all but the most successful cases, a standardized hash-access technique should be competitive with more special techniques. Moreover, the use of specially invented access techniques will often hide the algorithmic kernel of a program behind a distorting mass of accessing and filing procedures which grow to something much larger than the algorithm from which the program has been developed. The use within MIDL of a suitably devised standard hashing technique can avoid these difficulties, and allow MIDL programs to stay much closer to their SETL prototypes. For this reason, we shall provide standardized hashing primitives as a part of MIDL.

For this, we introduce an additional data object, the *maptable*, into MIDL. MIDL maptables, like the tables used in SETL to represent sets, will grow and shrink, probably by binary jumps, as functional values are added to and deleted from them. A maptable is capable of storing one of several functions of a bit-string argument; these functions may be bit-string or pointer valued.

To declare a maptable, we write

(23) DCL X MAP(argsiz,tp);

(as usual, the declarat on of several successive hashtables may be strung together). Here *argsiz*, a compile-time constant, denotes the size, in bits, of the intended argument to X; *tp*, a type name denoting the type of value V which X returns. (In effect, the value which X returns is as if declared by

(24) PTR X(tp) .

To retrieve a value from a maptable X, we write one of

(25a) X(s)

(25b) *field* X(s)

The form (25a) retrieves the 'entire' value V of X(s). The form (25b) retrieves the item pointed to by a field of X(s).

If accessed, undefined maptable entries are returned with omega in all pointer points and 0 in all bit positions. The diction

(26) .DEF. X(s)

returns 0 if X(s) is undefined, 1 if X(s) is defined.

The dictions (25a-b) may be used on the left-hand side of assignment statements, where they act either to define new maptable entries or to modify old ones. When a maptable entry is created by such an assignment, all the pointers contained in the new entry, with the possible exception of the very pointer which is the assignment target, are initialized to nil.

To drop a maptable entry, one writes

(27) .DROP. X(s);

To drop all entries in a maptable, one writes

(28) .DROP. X .

The following remark concerning implementation will clarify the semantics of maptables. A maptable is always accessed (in logically 'indirect' fashion) through an auxiliary pointer P stored in a single location; when the maptable grows and must be recopied, this pointer is changed, thus 'instantaneously' changing all other references to this table from the old to the new copy. If there exists only one program reference to the maptable, the pointer P can be stored in this location. If there exist several such references, and especially if a maptable can be accessed via many stored pointers, all of these pointers should point to the single pointer P. This adds an additional level of indirection in the access path leading to a particular table entry.

Note that we allow pointers to point at maptables, and allow maptables to be assigned. Therefore maptables act like quantiteis of type pointer. To give one quantity of type 'maptable' the same reference as another, we may simply write

(29) X = Y;

To cause a pointer field to point at a maptable, we write

(30) *field* V = Y .

For (22) to be used where Y is a maptable, the *field* etc. must have been declared as a maptable of similar type. A pointer field f_n in a structure is declared to point to a maptable by writing

(31) TYPE typename: f_1 pd_1, f_2 pd_2 ,..., f_n MAP(argsz, tp);

Here, the field *h* is a pointer to a maptable. *argsz* and *tp* are as in (23).

As in LITTLE, static variables may also be declared in a size or real statement. For example, the statements

```
REAL A,B,C;
SIZE D(WS), E(PS), F(PS);
```

have the same meaning as in LITTLE.

2. Dimensioned Subfields of Structures; Commonality Rules.

We allow the fields of a structured type (2) to be 'repeated', i.e. to be defined with dimensions. This is done by writing

(32a) TYPE typename: partname(n) typedesc, ...

in place of the simpler, undimensioned,

(32b) TYPE typename: partname typedesc, ...

In (32a), *n* is an integer constant denoting the number of times that the field *partname* is to be repeated. If a variable X is declared to be of a type (32a), an extra index is required in order to extract (or insert) elements of X. For example, in the presence of the declarations

(33a) TYPE WITHDATA: P PTR(WITHDATA), DATUM(3) REAL;

(33b) TYPE WITHARRAY: PTR(* WITHDATA) ,

(33c) DCL X WITHARRAY, ...;

the second entry in the 'DATUM' field of the J-th entry in the array to which X points is referenced by

(34a) DATUM(2) X(J) .

Adapting a useful syntactic convention from SIMULA 67, we now specify that in addition to *unprefixed* declarations of the form (2), MIDL will provide *prefixed* type declarations of the syntactic form

(35) TYPE typename: typename_0: partname_1 typedesc_1,...,
partname_n typedesc_n;

An example (cf. (3)) would be

(36) TYPE BIGNODE: LISTNODE: SUBLIST POINTER(LISTHEAD),
MEMBERNO BITS(10);

In (35), as in (2), *typename* is any admissible LITTLE name, which becomes the name of the type introduced by (35); $partname_1,\ldots,partname_n$ are valid names, all distinct, which come to name components of the type introduced by (35). Finally, the syntactic objects $typedesc_1,\ldots,typedesc_n$ are all type descriptors. A structure introduced by a prefixed declaration (35) inherits all the components of a structure of the prefixing typename_0 , plus the additional components $partname_1,\ldots,partname_n$ which appear explicitly in (35). Moreover, the new and the inherited components are arranged compatibly, so that expressions of the form

field obj

can be used for objects *obj* of type *typename*, whether *field* names an inherited component or a newly defined component of such an object.

3. Conversion between SETL and MIDL Object Forms. Access to SRTL Entries.

Under the present heading we shall propose conventions which secure two principal ends:

i. It should be possible to link MIDL to SETL so as to use SETL programs as frameworks within which developing MIDL programs can be debugged.

ii. It should be possible, by transcribing some 'innermost' part of a SETL program into MIDL, to produce a hybrid SETL/MIDL program which attains reasonable efficiency.

(a) Pointers as SETL Objects.

We allow MIDL objects declared as pointers to be members of SETL sets, and more generally to introduce such pointers as a new type of semantic object in SETL (as extended for communication with MIDL). This can be done with minimum modification to SRTL; pointers can be handled essentially as blank atoms, which are however flagged to show a different type. MIDL pointers differ from SETL blank atoms only in that field and indexing operators, i.e., constructions

(37) *index* pt and pt(*index*) , etc.,

may meaningfully be applied to them. Note that a pointer has a continuing identity irrespective of the values stored in the data object to which it points, e.g., an assignment

(38) *field* pt = 0;

does not either remove *pt* from a set from which it happens to be a member or insert *pt* into any other set.

(b) SETL Primitives in MIDL.

A MIDL variable or structure field may be declared to be a SETL data object by writing

(39a) DCL v SETLOBJ;

(39b) TYPE t:f SETLOBJ;

The following SETL constants are available in MIDL:

MIDL	SETLB	MEANING
.NL.	NL.	null set
.NULC.	NULC.	null character string
.NULT.	NULT.	null tuple
.TRUE.	T.	SETL true
.FALSE.	F.	SETL false
.OM.	OM.	undefined

Note that there is a distinction between SETL true and false and MIDL true and false. MIDL true and false are defined in the same way as in LITTLE.

Variables which are declared to be SETL objects may be used in standard algebraic expressions. The compiler, when it detects operands which are SETL objects, will compile a call to a routine in SRTL, which will perform type checking and call the appropriate routine for the operation. 'Mixed mode' expressions between SETL and MIDL operands are illegal and result in compile-time diagnostics.

Type constants are as follows:

MIDL	meaning
.INT.	integer
.BLANK.	blank atomic
.SET.	set
.TUPL.	tuple
.STR.	SETL character string
.LAB.	label
.BITS.	boolean string
.PTR.	pointer

The following operators which already are defined may be used with SETL objects:

OPERATOR	SRTL Routine Invoked
+	PLUS
-	MINUS
*	MULT
/	DIVIDE

[continued]

OPERATOR		SRTL Routine Invoked
=,	.EQ.	EQUAL
¬ =,	.NE.	EQUAL
< =,	.LE.	LE
> =,	.GE.	LE
< ,	.LT.	LT
> ,	.GT.	LT
¬ ,	.NOT.	BOOLNOT
∧ ,	.AND.	BOOLAND
∨ ,	.OR.	BOOLOR
	.EXOR.	BOOLEX
-	(unary)	PMINUS

A few remarks need to be made to clarify the semantics of comparison operations.

i. Comparison between two SETL object yields a MIDL true or false result.

ii. Since SETL is a value language, it is clear that if A and B are long SETL objects in the expression

(40) A .EQ. B

the values of objects A and B are compared to their full depth. However, if A and B are pointers to MIDL heap structures of the same type, the expression (40) results only in the comparison of the pointers to check whether they point to the same heap object. We therefore provide in the language the operations

(41a) P1 .EQL. P2

(41b) P_1 .NEQL. P2

where P1 and P2 are pointers. The result is true if the value of the object referenced by P1 is equal to the value of the object referenced by P2.

Other primitive SETL operations which are made available, generally in the form of infix or prefix operators, are listed below.

MIDL	SRTL Routine	Value Obtained
.NEWAT.	NEWAT	root word
EL .ELMT. E2	ELMT	1 or 0
.TYPE. E	TYPE	type constant
.NELT. E	NELT	MIDL integer
.ARB. E	ARB	root word
.DEC. E	DEC	root word
.OCT. E	OCT	root word
SETOF($E_1,\ldots,E_n$)		forms $\{E_1,\ldots,E_n\}$
TUPLOF($E_1,\ldots,E_n$)		forms $<E_1,\ldots,E_n>$

MIDL	
DIMINISHF(X,S); DIMINISHF($X_1,\ldots,X_n$,S);	SETL <u>lesf</u> and <u>lesfn</u> operations call *dimf*, *dimfaok*, or depending on the number and type of arguments.
F(X)	if F and X are SETL objects, will call either *of*, *ofbstr*, *ofcstr*, *oftuple*, or *ofset*. This operation is also available if F is declared to be a vector, bitstring, or character string; and X is a MIDL integer or bitstring.
F($X_1,\ldots,X_n$) F{X} F{$X_1,\ldots,X_n$} F[X] F[$X_1,\ldots,X_n$]	available if F and X are SETL objects; have the same meaning as do the corresponding SETL forms. We also allow these forms to be used in sinister position, to call the routines *sof*, *sofbstr*, *softupl*, *sofset*, *sofn*, *sofa*, *sofan*, *sofb*, *sofbn*.
.MIN. , .MAX.	for SETL objects, these call the SETL *min* and *max* library routines (infix).
.BOT. , .TOP.	the SETL <u>bot</u> and <u>top</u> functions (prefix)
.POW. , .NPOW.	the SETL 'pow' and 'npow' functions (.SPOW. is prefix, .SNPOW. binary) .

The .NELT. function may be applied to a MIDL pointer as well as a SETL object. It computes the dimension of an array object and the number of entires in a map table.

To add an element to a set or remove an element from a set we use the following statements, adapted from SETL:

(42a) E1 .WITH. E2; (AUGMENT)

(42b) E1 .LESS. E2; (LESS)

There is no implicit copying of E2 in dictions 42(a-b).

The operation F(X), where F is a *setlobj*, may be used in dexter and sinister positon. Note that if F is a vector, bit string or character string, one may not write F(3), for example, since 3 is a MIDL constant, not a SETL object. (Cf. the section on MIDL to SETL conversion operators, below.) We provide a function

(43) COPY(V)

which creates a copy of the heap object V and returns a pointer to the copy.

Here, V must be declared as a pointer or a *setlobj*.

Following the LITTLE style of extract operations and assignments, we also allow extraction to be performed on SETL bitstrings, character strings and tuples. This is provided by

(44a) .SUB. E1, E2, E3

(44b) .SUB. E1, E2, E3 = E4;

Diction (44a) results in a call to SRTL routine SUBSTR and (44b) to routine SSUBSTR. Operands E1,E2,E3, and E4 must be SETL objects. (More specifically, expressions E1 and E2 must evaluate to SETL integers, and E3 to a string or tuple.)

In order to make the SETL iterator accessible through MIDL, we make available an iteration header

(45) FOR E IN X;

Here X and E should have been declared as SETL objects. The scope of the iterator (45) is closed in the normal LITTLE style by

(46) END FOR;

The same diction is available with MIDL maps; the elements returned on successive iterations being the 'domain' elements of the 'pairs' implicitly stored by the map.

Input/output will apply to MIDL and SETL objects in syntactically similar forms but with different semantic implications. Two forms are provided: unformatted and formatted. The unformatted forms are (cf. LITTLE Newsletter 34, p. 9):

(47a) PUT filename var,var,...;

and

(47b) GET filename expn,expn,...;

Here *filename* names the source (or target) file to be used. The statements (47a), (47b) are intended to invoke binary input and output processes which are inverse to each other. To clarify the semantic intent of (47a,b), we must say something about the input and output of MIDL objects, which can, of course, contain pointers. When such an item X is written to a file, we proceed as follows: X and recursively all the items to which fields in X point, all items to which fields in these first items point, etc., are copied into a contiguous block of memory. This is a garbage-collector-like process. A binary copy of the resulting block of memory is then written out. The READ operator is then a function which takes the binary record generated by a write operation and brings it in, supplying an appropriate additive offset for each pointer field.

Formatted i/o is provided (as in LITTLE Newsletter 34) by GET and PUT statements with 'formatted output lists' (see NL 34, page 11). To allow SETL objects to be handled, we propose to introduce 'setlformat' as a new format type.

(c) Conversion Operators.

Certain of the atomic objects of MIDL can be converted to SETL objects, and vice versa. This applies to MIDL objects which are bitstrings, reals, and self-defining strings.

To make it possible for a programmer to call for these conversions, we provide a conversion operator in the two syntactic forms:

(48a) .CN. n, obj

(48b) .CN. setltype, obj

where *obj* is the object to be converted.
In (48a), *obj* is a SETL object which will be converted to a MIDL object. If *obj* is an integer or a bit string, the constant *n* specifies the number of bits of the resulting LITTLE bit string. If *obj* is a SETL character string, the result will be a self-defining string, and the constant *n* gives the maximum number of characters.

In (48b), *obj* is a LITTLE value, and it will be converted to a SETL object of type indicated by *setltype*. Setltype may be:

SETLINT	integer
SETLSTR	character string
SETLBSTR	bit string
SETLREAL	real number

If the result is to be a SETLSTR, *obj* must be a self-defining string. For example, the following expression yields a SETL character string:

(49) .CN. SETLSTR, 'this is an SDS'

4. Namescoping, Recursion, Parameter Transmission. ENTRY Variables.

The namescoping conventions in MIDL are modelled on LITTLE with the aims of preserving modularity and the ability to compile incrementally, and supporting recursion. Variables declared in the first routine are global, and, additionally, the NAMESET and ACCESS statements will be available.

All variables which are declared in routines after the first routine are local, except as declared global by use of NAMESET and ACCESS statements.

The EXTERNAL statement has the form

(50) EXTERNAL *name typedesc;*

and declares *name* to be the name of a function which is called from a LITTLE program but which will be supplied by the loader from a library that is not of the standard MIDL form. Here, *typedesc* defines the type of the value which is returned by the external function *name*. The purpose of this statement is to make it possible to link routines written in FORTRAN etc. with MIDL programs. This involves supplying the MIDL compiler with the information it needs to compile correct code. Of course, it is assumed that loader supplied external routines use calling sequences which at the machine level are compatible with those generated by the MIDL compiler.

To match the SETL semantics we provide recursive routines, which are handled in an essentially conventional way.

A routine or function which is used recursively must be declared in either of the forms:

(51a) SUBR name RECURSIVE;

(51b) FNCT name RECURSIVE;

When a recursive routine is entered, the code address of the call and the recursive variables are stacked. When returning from a routine declared recursive, the return address is then obtained from the stack, and not from the entry point. We allow the keyword STACK to be appended to the declaration of variables known in a routine. The value of each such variable will be transferred to the system stack when the routine is entered at a level of recursion greater than 1, and will be restored from this stack when return is made from the routine. For local variables of recursive routines, STACK is the default, and the appended keyword NOSTACK may be used to suppress stacking.

In part because the semantics of SETL procedures cannot readily be mimicked in their absence, we provide objects and variables of the type ENTRY in MIDL. An ENTRY object is created each time a procedure is compiled by the MIDL compiler. We may think of the compilation of such a procedure as initializing a variable of type ENTRY, whose symbolic name is that of the SUBR or FNCT which is compiled. However, names declared in this implicit way behave in a somewhat special manner as 'entry constants', which cannot properly be assignment targets. The fact that the values of such names are invariant is exploited to allow the generation of efficient linkages when these names appear in function or subroutine calls. In addition to these 'entry constants', we also allow entry variables, which are declared in the form

(53) DCL *varname* ENTRY

for subroutines and also for function-variables which may return a value of one of several types. For function variables which return a value of fixed type we provide the declaration

(54) DCL *varname* ENTRY *typedesc* .

More generally, ENTRY and ENTRY *typedesc* are allowed as type descriptors. An ENTRY variable, and more generally a field or array component of type ENTRY, can be the target of an assignment whose right-hand side is an entry constant, entry variable, and in general any entry quantity.

In compiling a call to an entry constant we generate information which will cause the laoder to set up a fixed linkage to a known object (of type 'procedure') when an executable, closed package of subroutines is being built. A call involving an entry variable will be compiled differently, with code which sets up a parameter list and then transfers to an entry address which will have been transmitted dynamically rather than being supplied statically by the loader.

When compiling a call to a function, the MIDL compiler requires information concerning the type of value returned by the function. The ENTRY descriptors defined above serve in the case of calls to variable procedures to make this information available. For corresponding use when we deal with procedure constants, we provide a statement of the form

(54) EXPECT *function-name (typedesc);*

where *function-name* is the name of a function whose definition will follow its first use, and where *typedesc* defines the type of value which this function returns. If the definition of a function precedes its first use, no EXPECT statement is necessary.

For an initial implementation of MIDL, we propose to generate LITTLE source code. The code output will be similar to that generated by the SETL translator system (SETLBEAST). This will avoid adding an extra layer of complexity to the LITTLE compiler, which is currently being worked on by several people. Furthermore, the source may be undergoing major modifications in order to install global optimizations. At a later time a suitable intermediate language should be implemented as the target of both the translator system and MIDL.

The decision to generate LITTLE as target code from MIDL will probably require some changes to the LITTLE compiler. There is, in particular, a problem with the current parameter-passing convention. Currently, if the actual parameter is simple, its address is passed. However, if the parameter is indexed, the value is passed. This convention has proved to be somewhat awkward in that in order to modify an array location which is a parameter, the base array address and the index must be transmitted as 2 separate parameters. In the LITTLE code generated from MIDL, this will be more of a problem, since all variables which are declared as pointers will be compiled into indexed expressions of the form

STORAGE(I) .

LITTLE 37-23

Suppose a pointer variable V, which is a SETL object, is passed as a parameter. If V is a short object, the value of the item is passed, and an assignment to the parameter within the called routine will not change the value of the variable. If V is a long object, however, the heap address will be passed, and an assignment to the parameter will change the value of V. The conventions of LITTLE should be changed to avoid such inconsistencies.

Appendix. MIDL BNF Grammar (without I/O).

In the grammar below, an asterisk following a metavariable name means the item may appear 0 or more times. An asterisk preceding a name designates a lexical token type. The lexical types are:

<*name>	LITTLE identifier name
<*compname>	component name: LITTLE identifier
<*const>	constant
<*notsemicolon>	any token but ';'
<*binop>	binary operator
<*unop>	unary operator

(1) <program> → <routine> <routine*>

(2) <routine> → <routhdr><block> <ender>

(3) <block> → <labstatement> <labstatement*>

(4) <labstatement> → /<*name> / <statement>

→ <statement>

(5) <statement> → <declstat>

→ <compstat>

→ <simplifstat>

→ <simplstat>

(6) <routhdr> → <rhdr>;

→ <rhdr> RECURSIVE;

(7) <rhdr> → SUBR <*name>

→ SUBR <*name> <arglist>

→ FNCT <*name> <arglist>

```
(8)   <declstat> → <declaration> STACK;
                 → <declaration> UNSTACK;
                 → <declaration>;
                 →  ACCESS <*name> <cname*>;

(8a)             →  DATA <dataspec> <coldataspec*>;
                 →  DIMS <attrspec> <cattrspec*>;
                 →  EXPECT <vardcl> <cvardcl*>
                 →  EXTERNAL <vardcl> <cvardcl*>;
                 → <typedef>

(9)   <declaration> → SIZE <attrspec> <cattrspec*>
                    → REAL <*name> <cname*>
                    → DCL <vardcl> <cvardcl*>

(10)  <vardcl>   → <*name> <typedesc>

(11)  <typedesc> → <*name>
                 → <typexpr>

(12)  <typexpr>  →  PTR
                 →  PTR (<typedesc>)
                 →  BITS (<constexpr>)
                 →  REAL
                 →  SETLOBJ
                 →  ENTRY
                 →  ENTRY <typedesc>
                 →  MAP (<constexpr>, <typedesc>)
                 →  PTR (*<typedesc>)

(13)  <cvardcl>  →  , <vardcl>

(14)  <typedef>  →  TYPE <typename> <tdescpart>;

(15)  <typename> → <*name>:

(16)  <tdescpart> → <typename*> <tdescsp> <ctdescsp*>
                  → <typexpr>

(17)  <tdescsp>  → <*compname> <typexpr>
                 → <*compname> (<constexpr>) <typexpr>
```

```
(18)  <ctdescsp> → , <tdescsp>

(19)  <attrspec> → <*name> (<constexpr>)

(20)  <cattrspec> → ,<attrspec>

(21)  <dataspec> → <*name> (<constexpr>)  =  <dataval*>

                 → <*name>  =  <dataval*>

(22)  <dataval>  → <dataexpr> <cdataexpr*>

(23)  <cdataexpr> → , <dataexpr>

(24)  <coldataspec> → : <dataspec>

(25)  <dataexpr> → <constexpr> (<constexpr>)

                 → <constexpr>

(26)  <compstat> → <opener> <block> <ender>

(27)  <opener> → NAMESET <*name>;
               → WHILE <expr>;
               → UNTIL <expr>;
               → DO <*name> = <expr> TO <expr> BY -<expr>;
               → DO <*name> = <expr> TO <expr> BY  <expr>;
               → DO <*name> = <expr> TO <expr>;
               → IF <expr> THEN <block>  ELSE
               → IF <expr> THEN
               → FOR <expr> IN <expr>;

(28)  <ender> →  END <notsemi*>;

(29)  <notsemi> → <*notsemicolon>

(30)  <simplifstat> → IF <expr> <simplstat>

(31)  <simplstat> → CALL <*name> (<expr><cexpr*>);
                  → CALL <*name>;
                  → CONT <notsemi*>;
                  → GOTO <*name>;
                  → GOBY (<*expr>) (<*name> <cname*>);
                  → GOBY <*name> (<*name> <cname*>);
```

```
                              → QUIT <notsemi*>;
                              → RETURN;
                              → .DROP. <expr> (<expr>);
                              → .DROP. <expr>;
                              → <expr>  .IN.  <expr>;
                              → <expr>  .OUT. <expr>;
                              → <assignstat>
(32)  <assignstat>            → <lhside>  =  <expr>;
                              → <setaccs> = <expr>;
(33)  <cexpr>                 → , <expr>
(34)  <arglist>               → (<*name> <cname*>)
(35)  <cname>                 → , <*name>
(36)  <lhside>                → <extbeg> <lhside>
                              → (<lhside>)
                              → <vatom>
(37)  <expr>                  → <expr> <*binop> <expr>
                              → <term>
(38)  <term>                  → <exprbeg*> <atom>
(39)  <exprbeg>               → <unop>
                              → <extbeg>
(40)  <unop>                  → <*unop>
                              → .CN. <cndesc>, <expr>,
(41)  <extbeg>                → .F. <expr>, <expr>,
                              → .S. <expr>, <expr>,
                              → .E. <expr>, <expr>,
                              → .SUB. <expr>, <expr>,
                              → .CH. <expr>,
                              → <*compname>
                              → <*compname> (<constexpr>)
                              → <*name>::
                              → ↑
```

```
(42)   <cndesc>   →  BITS(n)
                  →  CHARS(n)
                  →  PTR
                  →  SETLINT
                  →  SETLSTR
                  →  SETLBSTR
                  →  SETLREAL
                  →  SETLPTR
(43)   <atom>     →  (<expr>)
                  →  <vatom>
                  →  .DEF. <*name> (<expr>)
                  →  .DEF. <*name>

(43a)             → <*const>
                  → <setaccs>

(44)   <setaccs>  →  <*name> (<expr> <cexpr> <cexpr*>)
                  →  <*name> [<expr> <cexpr*>]
                  →  <*name>{<expr> <cexpr*>}

(45)   <vatom>    →  <*name>
                  →  <*name> (<expr>)
```

COMMENTS:

(8a) Variables appearing in DATA statements must be static. For example,

```
DCL V BITS (83);   DATA V = 17;
```

is valid, while

```
DCL V PTR (heapobj);   DATA V = 7;
```

is illegal.

(17) A *compname* is any valid LITTLE name. A component name may be used to name a component of one or more structures, but the name may not be used to name anything else (e.g., variable, subr).
Except for component names, all identifiers must be unique.

(19) <constexpr> is a constant expression; i.e. an arithmetic expression which must evaluate at compile time to a constant.

(32) The following are examples of valid assignments:

```
compname1 compname2 V = E;
compname1 (compname2 V(I)) = E;
compname1 ↑ compname2 ↑ V = E;
compname ↑ ↑ V = E;     /* two dereferences */
↑ tp :: V = E;          /* tp qualifies V */
tp :: ↑ V = E;          /* tp qualifies object
                           referenced by V */
tp1 :: tp2 :: V = E;    /* tp1 qualifies V.
                           tp2 ignored. */
```

The following are illegal:

```
compname F{X} = E;
compname F[X] = E;
compname F(X1,X2,...,XN) = E;
```

(37) Binary operators are listed below with operand type and a number indicating operator precedence strength. The stronger the operator strength, the higher the number.

Operand types are coded as follows:

M MIDL primitive object (pointer)
L - LITTLE object (bit string)
L(SDS) - self-defining string
S - SETL object

operator	precedence	operand1	operand2	result
.C.	1	L	L	L
.CC.	1	L (SDS)	L (SDS)	L (SDS)
.OR., v	2	LS	LS	LS

(continued)...

operator	precedence	operand1	operand2	result
.EX., .EXOR.	2	LS	LS	LS
.AND., .A., ∨	3	LS	LS	LS
.EQ., =	4	LMS	LMS	L
.NE., ¬ =	4	LMS	LMS	L
.EQL., .NEQL.	4	LMS	LMS	L
.LE., <=	4	LS	LS	L
.GE., >=	4	LS	LS	L
.GT., >	4	LS	LS	L
.LT., <	4	LS	LS	L
+	5	LS	LS	LS
-	5	LS	LS	LS
*	6	LS	LS	LS
/	6	LS	LS	LS
.INS.	7	L (SDS)	L (SDS)	L
.ELMT.	4	S	S	L

(40) Unary operators are:

operator	operand	result
¬, .NOT., .N.	SL	L
-	SL	SL
.NB.	L	L
.FB.	L	L
.TYPE.	SM	L
.NELT.	SM	L
.ARB.	S	S
.DEC.	S	S
.OCT.	S	S
.MIN.	S	S
.MAX.	S	S
.BOT.	S	S
.TOP.	S	S
.POW.	S	S
.NPOW.	S	S

(43a) Constants include the type constants:

.INT.	.STR.
.BLANK.	.LAB.
.SET.	.BITS.
.TUPL.	.PTR.

and other SETL constants:

.NL.	.TRUE.
.NULC.	.FALSE.
.NULT.	.OM.

Additionally, the following are reserved words, which are system function names:

NEW	COPY	SETOF
TRIM	DIMF	TUPLOF