MIDL: A Hybrid Language of Medium Level.

by

E. Deak, M. Shimasaki*, and J. Schwartz

Computer Science Department,

Courant Institute of Mathematical Sciences

New York University

The MIDL language currently being developed at NYU incorporates features of two other NYU languages (SETL and LITTLE), and welds these rather different languages together. The SETL language (cf. Kennedy and Schwartz [3]) is a very high level algorithm specification language with sets and tuples as its data types; SETL supports many set theoretic dictions. Because of SETL's high level, it is essential that it be implemented in an efficient systems-oriented language. We also felt it to be essential that SETL be portable between machines.

These goals shaped the design of the implementation language used to realise SETL: this is LITTLE (cf. Shields [6]) a FORTRAN-like, machine independent language. The LITTLE compiler is itself written in LITTLE, and is carried to a new machine by a bootstrap procedure. The extensive run-time support library required by SETL is written entirely in LITTLE. LITTLE achieves efficiency comparable to that of standard FORTRAN compilers.

SETL serves well for the specification of complex algorithms, but not for writing production software. The present SETL implementation, which is fully compiled but not globally optimized, attains between 1/6 and 1/50 of the speed of FORTRAN (depending on the nature, combinatorial or arithmetic) of the program being run. We expect that global optimization will improve SETL's efficiency to lie between 1/2 and 1/10 of that of FORTRAN, and have begun to develop a global SETL optimizer (see Schwartz [4,5]

* On leave from Department of Computer Science, Kyoto University

† Work supported by the National Science Foundation, Office of Computer Activities, Grant DCR75-09218.

for some of the optimization approaches which will be used.) This optimizer will itself be a large complex program. It is being specified in SETL, but a production version of the optimizer will also be required as a component of a new SETL system. The optimizer is complicated enough to make dynamic storage a highly desirable feature of the language used to implement it.

These considerations led us to propose the MIDL language as a major extension to LITTLE. Efficiency requirements keep MIDL reasonably close to the low level semantic approach of LITTLE; but MIDL runs in a garbage-collected memory millieu fully compatible with that of SETL. This makes SETL primitives available for use in MIDL, and, more significantly, allows MIDL routines to be called from SETL. Thus one can use a SETL program P as a framework within which developing MIDL programs can be debugged. It is also possible, by transcribing some innermost part of P into MIDL, to produce a high efficiency version of P,much as is done when FORTRAN inner loops are replaced by assembly language code. Thus MIDL will serve to bring programs originally written in SETL to production efficiency levels,by hand transcription. MIDL is also upwardly compatible from LITTLE, and any LITTLE program is a valid MIDL program. A programmer can therefore improve the efficiency of a MIDL program by rewriting sections of it in pure LITTLE.

In implementing MIDL, we have tried as far as possible to preserve the machine independence which characterizes LITTLE and SETL. This entails avoiding word-length dependencies and explicit restrictions on the number and position of pointers in a machine word.

Description of the MIDL Language

The syntax of MIDL is very much like that of LITTLE.

A program consists of a sequence of subroutines and functions. Program text may be free-format; statements are terminated by semi-colons,; standard compound statement forms are provided to support structured programming style. A simple macro-processor allowing parameter substitution is provided; this has proven to be an invaluable tool for writing clear, portable programs.

All variables must be declared. The namescoping scheme is static. Variables are by default local to a procedure but may be made available to other routines by declaring them within the scope of a *nameset*. A nameset may be accessed by another routine via an *access* statement; all variable names in the nameset then become known to the accessing routine.

Data Types

LITTLE supports no data type * notion. Data objects are bitstrings or one-dimensional arrays of bitstrings. Declarations specify the size (in bits) and dimension of variables. Field extraction operations access subparts of bitstrings.

MIDL supports several primitive data types going beyond the fundamental bitstring of LITTLE. The predefined types are: bitstring, real number,pointer, SETL object, entry object, and maptable. These will be described more fully below. The MIDL user may define new types, which are always structures consisting of one or more components, each component being of a specified type. For an operation to be valid, the declared types of the operands must be (statically) acceptable to the operator appearing in the operation. Note that this static treatment of data types contrasts with the totally dynamic treatment of types in SETL.

* With the grudging exception of real numbers.

Variable are declared in the form

DCL $name_1$ $typedesc_1$, ..., $name_n$ $typedesc_n$;

Here $name_i$ names a variable and $typedesc_i$ is a type descriptor. A type descriptor may either be a user type name, indicating that the variable is of user defined type, or is one of the following:

(a)	BITS(n)	A bitstring of size *n*.
(b)	REAL	Real number (implementation dependent)
(c)	PTR(typename)	A pointer to a structure.
(d)	PTR(*typename)	A pointer to an array of structures.
(e)	SETLOBJ	A SETL object.
(f)	MAP(n, typename)	A maptable defining a function from bitstring arguments of size *n* into structures of type *typename*.
(g)	ENTRY	A procedure entry variable.

New types (structures) are introduced by type definitions of the form

TYPE typename: $cname_1$ $ctypedesc_1$,
$cname_2$ $ctypedesc_2$,
. . .
$cname_n$ $ctypedesc_n$;

The name of the new type is *typename*; $cname_i$ and $ctypedesc_i$ specify the name and type of its i-th component.

An example would be

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

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

cname V

if V points to a non-array structure, or (for a vector component) writes

V(index)

if V points to an array, or writes

partname V(index).

in case of an array of structures.

The diction

↑ V

accesses the whole of a non-array object pointed to by a pointer V.

Storage for structures which are accessed via pointers must be explicitly allocated. A structure of type *t* is created by a function call of the form

NEW(t).

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

NEW(t,n).

There is no explicit deallocation; free storage is recovered by means of garbage collection.

Maptables

The array notion is well adapted to the representation of functions defined on a dense range of integers, but is not adequate when we attempt to deal with functions defined on a sparse range of integers. In SETL such functions raise no problem, since SETL's general'mapping' concept handles sparsely defined functions well; the technique used is hashing. When faced with a sparsely defined map, a programmer striving for efficiency in a low-level language will often invent *ad hoc* encodings or data arrangements which expedite access to map values. These encodings often hide the algorithmic kernel of a program behind a distorting mass of accessing and filing procedures, which can grow to be something much larger than the algorithm from which the program has been developed. In most cases a standardized hash-access technique will be competitive with more special techniques;

recognising this, MIDL supports a standardized hashing technique through its *maptable* notion. A MIDL maptable is capable of storing functions of a bitstring argument; the value of the function can be an
object of any user-defined type. MIDL maptables, like the tables used in SETL to represent sets, grow and shrink as functional values are added to and deleted from them.

To declare a maptable, we write

```
DCL x MAP(argsize, type);
```

Here, *argsize*, a constant, denotes the size (in bits) of the argument which will be supplied to x; *type* is a type name denoting the type of value which x returns. To retrieve (resp. store) a value from (resp. into) a maptable, we write

```
x(s)          resp.          x(s) = val;
```

where s is a bitstring of size *argsize*.

Interface between SETL and MIDL

The following MIDL features support communication between SETL and MIDL:

(1) *SETL object* is a primitive data type in MIDL. A variable is declared to be a SETL object by writing

```
DCL  x    SETLOBJ;
```

Every SETL operation has a counterpart in MIDL; these have the semantics of corresponding SETL operations. For example, in SETL, if variables A and B are sets, the expression

```
A + B
```

yields the set union of A and B; the same is true in MIDL if A and B are declared to be SETL objects.

'Mixed mode' expressions involving both SETL and MIDL operands are illegal and produce compile-time diagnostics. However, the MIDL compiler makes no distinction among the various SETL primitive objects (e.g.sets,tuples, integers); run-time type checking is performed as in SETL.

A SETL algorithm can be improved in efficiency by keeping part of it in SETL but transcribing critical procedures into MIDL. The SETL algorithm may also serve as a specification for a more efficient version written wholly in MIDL. In developing a MIDL code, SETL objects in its original specification can be selectively and gradually translated into MIDL structures and maptables, or can be left as SETL objects. This gives a programmer control over the semantic level of his program. In general, the lower the semantic level chosen, the more efficient the resulting program.

(2) We allow MIDL pointers to be members of SETL sets; SETL treats these objects as a new type of blank atom.

(3) MIDL provides conversion operators which transform certain of the MIDL atomic objects to SETL objects, and vice versa. For example, to convert a MIDL bitstring *b* into a SETL integer, one can write

```
.CN.  SETLINT,  b
```

Thus if *a* is declared to be a SETL object, the expression

```
a +  .CN. SETLINT, b
```

is valid; the result will be a SETL object.

MIDL extends the subroutine linkages of LITTLE, to come to a closer match with SETL semantics. Recursion is supported in the conventional way. A routine which is used recursively must be declared in the form

```
SUBR  name  RECURSIVE;
```

The compiler can therefore distinguish between recursive and non-recursive routines and need not generate the more costly prologues and epilogues required by recursion in cases when these are not necessary.

The parameter passing mechanisms of MIDL and LITTLE are the same. The address of an actual parameter is passed if the parameter is a simple variable; otherwise, the address of a temporary is passed.

MIDL also provides objects and variables of type *entry*; these correspond to the procedure objects of SETL. Entry variables can be called in the same way as (constant) subprocedures.

Run-Time Environment

The run-time environment of MIDL incorporates the run-time environment of SETL (see [4]). All storage is divided into a STACK area and HEAP area. Pointers reference locations in the HEAP area; the heap is managed by a compacting garbage collector. The compiler allocates a stack location to every variable which stores a pointer. A second stack called RSTACK is used along with STACK to implement recursion; RSTACK stores recursive variables and parameters not involving pointers, and therefore is not involved in garbage collection.

Variables are treated as'static' in the PL/I sense , except that storage for local variables of recursive routines is re-allocated each time the routine is invoked and deallocated upon return.

Each MIDL routine has associated with it a base environment block, which is a block of consecutive STACK locations reserved for local variables. A block of STACK words is also associated with each global nameset. The sizes of all such blocks are known at compile time; however, in order to make routines separately compilable, these blocks are not allocated until run-time. Instead, for each block the compiler generates a variable which, after initialisation at the start of execution, references the beginning of the block.

References to a variable belonging to the block are then compiled as an offset from this base pointer.

Non-recursion related requests for dynamic storage are fulfilled from the HEAP. In MIDL such requests appear as uses of the NEW function, which returns a pointer to the allocated block.

Structures

A MIDL structure is represented as a block of one or more words; structure components are packed when appropriate. A structure may or may not contain pointers to objects in the heap. Structure words containing pointers must be in a garbage-collector compatible 'STACK word format'. An array of structures is represented as a contiguous block of scalar structures.

The compiler computes an internal representation for each structure defined in a program. This template is computed at the time the associated structure type declaration is processed; thus subfield address offsets are computable at compile time. References to MIDL structures and structure components generate in-line code. Operations involving SETL objects are compiled into calls to run-time routines of the SETL run-time library.

Maptables

Maptables, like the tables used in SETL to represent sets, are stored as hash tables. They grow and shrink by binary jumps as values are added to and deleted from them.

A maptable is always accessed through an auxiliary pointer P; when the maptable grows and must be recopied, the pointer is changed, thus instantaneously updating all references to the table. This use of an auxiliary pointer adds an additional level of indirection in the access path which leads to a particular table entry.

The index used to enter a maptable is computed from a bitstring argument of a fixed, declared number of bits. Entries which hash to the same position are chained together. When undefined entries are accessed, a copy of the maptable value template is initialized and returned. A small package of run-time routines provides the various hashing operations needed to support the maptable construct.

Implementation of the MIDL Compiler

In developing the MIDL compiler a 'two stage programming' approach was used. Before the actual coding was begun, a non-executable specification for most of the compiler was written in SETL. This specification served subsequently as documentation for the production compiler. The semantic power of SETL allows succint expression and frees the programmer from concerns about details of data structures, allowing concentration on design issues. We hoped that production implementation would then proceed in an orderly, straightfoward, and organized way.

This expectation has been borne out. The SETL specification defines the procedural structure of the compiler; various decisions relating to efficiency, bookeeping, data structures etc., are then faced just before actual coding begins, at which time these lower-level considerations can be better handled, since the basic organization of the compiler has already been determined.

Typical issues which must be handled as the compiler is reworked in LITTLE are static table overflows, hashtable management, and the field size and formats of table entries. These issues do not arise in SETL, which provides dynamically expandable data structures, sets usable as mappings, and recursion which can be used advantageously to describe parse tree handling.

In the production version of the MIDL compiler we were able to re-use the front end of the LITTLE compiler. Consequently, the MIDL compiler has an overall structure much like that of the LITTLE compiler. Both compilers consists of three separate overlays.

The first overlay performs lexical processing and macro expansion. The second overlay parses and constructs intermediate tables. This overlay also performs all type checking and emits error diagnostics if necessary. The parsing technique used is topdown-advancing. The parser is driven by tables produced using a meta-compiler system, whose ultimate input is a suitably extended BNF grammar.

The third overlay of the MIDL compiler generates target code from the intermediate table produced by the second overlay. The target language of the MIDL compiler is LITTLE itself.

This approach minimises both implementation and debugging effort. It also ensures compatibility with the existing SETL run-time library. Finally, it ensures that MIDL will be as transportable as LITTLE. A disadvantage of using LITTLE as a target language is of course the expense of compiling the LITTLE code produced by the MIDL compiler. However, the LITTLE compiler is rather fast (it compiles 6000 cards/minute on the CDC 6600) and produces good code. To speed up recompilation, character handling can be bypassed, and token streams passed directly.

Conclusions

Because the production version of the MIDL compiler is not yet complete, it is premature to draw conclusions about MIDL's success in regard to usefulness and efficiency. However, we very much expect that MIDL will be a suitable tool for developing a production version of a global SETL optimizer from the (already formidable) SETL version of this optimizer. More generally, it will make SETL more widely useable by allowing critical sections of SETL programs to be optimized manually. PL/I and ALGOL 68 are existing languages with roughly the same semantic level as MIDL; However, MIDL is compatible with our existing SETL software, and is highly transportable.

Because MIDL is a hybrid of SETL and LITTLE, its design and implementation did not proceed from scratch, but were dictated by the existing features of LITTLE and of the run-time library of SETL. The two pass programming technique we have used has proved to be successful: the production compiler code is very close to the original SETL specification, and the SETL version is indeed a useful document for understanding the production version.

References

1. J. Cocke and J. T. Schwartz, Programming Languages and their Compilers. Lecture Notes, Computer Science Dept., Courant Institute of Mathematical Science (1970).
2. K. Jenson and N.Wirth, PASCAL: User Manual and Report. Springer Publishing Company, (1974).
3. K. Kennedy and J.T. Schwartz, *An Introduction to the Set Theoretic Language SETL*. Computers & Mathematics with Applications, vol. 1, pp. 97-119. Pergamon Press (1975).
4. J. T. Schwartz, On Programming: An Interim Report on the SETL Project. Installment 1 - *Generalities*. Installment II - *The SETL Language and Examples of its Use*. Computer Science Department Courant Institute of Mathematical Sciences (1973).
5. J.T. Schwartz, *Optimization of Very High Level Language I. Value Transmission and its Corollaries*. Journal of Computer Languages, vol. 1, # 2, pp. 161-194 (June 1975). II. *Deducing Relations of Inclusion and Membership*. Journal of Computer Languages, vol. 1, # 3 (1975).
6. D. Shields, Guide to the LITTLE Language. *LITTLE Newsletter # 33*, (March 1974).