SETL Newsletter # 169 April 11, 1976

Some Changes to the SETL Language
in Preparation for the Optimizer Implementation

Robert Dewar
Art Grand
Len Vanek
Ed Schonberg

I. Introduction

In the course of designing the SETL optimizer we have decided on a number of changes to the language itself. Several changes affect the basic semantics of SETL, while the remainder are syntactic and lexical.

II. Semantic Changes

The following semantic changes have been made to SETL.

1. Program Structure:

A program consists of a set of separately compiled *modules*, each of which contains a set of functions and subroutines. Variables are local by default but may be declared global to a module. Global variables may be made *public*, allowing them to be *included* by other modules. The user may select which variables are stored statically and which are stacked on entry to a routine.

Every program must contain a module called '*main*'. This module should contain a block of code before the first procedure, which is treated as a main program.

The syntax for modules and declarations is given at the end of Section III.

2. Parameter Passage:

SETL allows two mechanisms for parameter passing: call by value and call by value with delayed value return. All parameters of functions are called by value. However for subroutines the user can decide which parameters will have value return. This decision must be indicated both in the *define* statement and in the call statement. In either case value return is indicated by suffixing each affected parameter with an equal sign.

3. Functions, Labels, and Subroutines:

are treated as constants and may not be redefined. Labels may appear anywhere within a routine.

4. A goto:

can only branch to a label in the currently active procedure; it cannot branch into the middle of a loop.

5. Maps:

are sets of pairs. Multivariate maps are described by sets of pairs whose second components are pairs, and not as sets of longer tuples, as was previously the case. See SETL Newsletter 166 for details.

6. Several operators have been redefined:

a. * means boolean and when applied to bit strings

b. + means boolean or for bit strings

c. / means exclusive or for bit strings

d. || means bit, character and tuple concatenation.

e. x subset y means y incs x

f. and and or apply only to bit strings of length 1. They evaluate their second argument only when necessary.

g. ** means exponentiation.

The precedence of operators has been greatly revised. A list of operators and their precedences is given at the end of this newsletter.

7. Order of Evaluation:

The order of evaluation of expressions is defined only for the is operator. If a variable is the target of an is and appears elsewhere in the same expression, the variable will be treated as if evaluation were left to right. The right-hand side of an assignment is evaluated before the left-hand side.

8. Sinister assignments:

Partially generated sinister assignments (as described in O.P. p. 181) are provided. Value-receiving expressions have the following

syntax:
<vexpr> = <name><index>[<index>...]
where <index> is a bracketed argument list, i.e.:
(x,y,...) or {z,t,...} or [S,T,...]
General multiple assignments are described by tuples of value-receiving expressions, e.g.
<f(x),g(y),h{z,t},<x(2), x(1)>>
Some components of such a tuple may be replaced by a dash, indicating a dummy assignment to that component. E.g.
<x,-,y> = z; is equivalent to
x = z(1);
y = z(3);

9. Initialization:

A simple form of the *initially* block has been added. Only *static* variables can be initialized; this initialization is performed before the start of execution.

10. Quit and continue:

The full *quit* and *continue* statements have been implemented. A *quit* or *continue* statement is applied to the innermost loop whose opening tokens match the tokens following the keyword quit or continue.

A *quit* or *continue* statement applied to a multiple iterator such as

(∀x ∈ s, y ∈ ss)

must act on the outermost loop.

11. An *elseif* statement has been added.

This allows the construction of-else-blocks which start with -if-. For example:

```
if c1 then
   block1
else
  if c2 then block2 end;
     block3
  end;
```

$block_3$ will be executed if c_1 is false, regardless of the truth-value of c_2.

12. The forms s(I:) = y and y = S(I:) have been implemented. Substring assignments may replace a substring of one length with a string of another length.

13. A stop statement has been added.

III. Syntactic Changes

Several new syntactic forms have been added. In most cases they can be thought of as macros for the standard constructs.

1. Iterators:

a. The set theoretic iterator can have a general left-hand side as its first argument, e.g.,

```
(∀<x,y> ∈ s)
```

This is a macro for

```
(∀t ∈ s)
   <x,y> = t;
```

Like all multiple assignments, this will cause an abort if t is not a tuple. This iterator may be used only for sets.

b. A new iterator form for tuples and maps is provided. It has the following syntax

∀ <left-hand side> = <map exp><index>[<index>...]

where <mapexp> is either a name or a parenthesized expression which evaluates to a map or tuple. <index> is either (x,y...) or {x,y...}. For example

```
(∀y = f(x))                    (1)
(∀<y,z> = f{x}(t))
(∀g(x) = (t(2))(x))
```

if f is a map, then (1) is equivalent to

```
(∀x ∈ domain (f))  y = f(x);...
```

if f is a tuple, then it is equivalent to

```
(∀1< = x< = #f)  y = f(x);...
```

2. The iterator statement, the existential quantifier and the universal quantifier may begin with a list of bound variables. The list is optional; however if it appears it must contain all the bound variables in the order in which they appear in iterators.
Some examples are:

```
(∀x, 1 <= x <= 100)
∃y, x, y = f(x) | c(x,y)
```

3. Set formers:
Three types of expressions may appear in set formers:

a. A single iterator. For example

```
{x ∈ s | c(x)}
{1 <= j <= n}
```

Here the set will contain all the values of the bound variable.

b. An expression followed by a series of iterators i.e.
{<x, <y, z>>, x ∈ s, <y, z> ∈ sl}

c. A list of expressions.

4. Tuple formers may contain the same expressions as set formers. E.g. <e(x), x ∈ s|c(x)>.

5. The membership operator is written in. Its complement is notin.

6. Functions and operators may be applied over the range of a set. Examples:

```
f[x]      means  {f(t), t ∈ x}
dec [x]   means  {dec(t), t ∈ x}
[A] + 1   means  {t + 1, t ∈ A}
```

7. Calls with no arguments must contain a null argument list:
x = dummy ( );

8. Assignments of the form

a = a op ... ;

may be abbreviated

a op ... ;

For example

(1) a + 1;
(2 x with y;
(3) x less y;

The in and out statements are replaced with (2) and (3) above.

The syntax of the from statement:

e from s ;

is retained. However, e can be a general value-receiving expression; s must be a simple name.

9. Code blocks have been removed from the language. The operators hd and tl have also been removed.

10. Declarations

Declaratory statements serve 3 purposes:

a) They establish the scope of program variables

b) They specify the storage class of program variables.

c) They assign modes to them and describe structural (basing) relations among them.

There are 4 declaratory statements: module, declare, external, and mode.

i. Each module is bracketed by the statements:

module *module-name* ;

and

finish;

ii. declare statements appear at the beginning of a module, before procedure definitions or executable code; or at the beginning of a procedure definition. The variables being declared are global to the module in the first case, local to the procedure in the second.

The declare statement has the form

declare <dlist> [,<dlist>...];

where

<dlist> → <name> [,<name>...] <options>

The options specify scope, storage class and basing mode of variables. For the first two options, the following attributes can be specified:

a) public: makes the variables in the <dlist> public

b) stack(<rname>): The variables in the <dlist> are stacked each time the routine *rname* is entered.

c) static variables are stored statically.

The options for local variables are

a) stack variable is stacked whenever its routine is entered

b) static

The stack and static options are mutually exclusive. The default is static for global variables and stack for local variables.

Example:

```
declare  x, y, z  static public,
         x1,x2 stack (proc1),
         s1,s2;              /*s1,s2 are global
                               and static*/
```

basing options are described in detail in the following section.

iii. The external statement gives one module access to public variables from another module. The included variables may be used under an alias. The syntax of the external statement is:

<external> → external <epart> [,<epart>...]

<epart> → (module name) <aliased namelist> <basing options>

The components of an aliased namelist are either names, or parenthesized pairs (global name:aliased name)

Example:

```
external (lib1) (x:lib1x), y, z
         (lib2) (x:lib2x), (fun:lib2fun);
```

This statement makes variables x, y, and z from module *lib1* accessible within the current module; y and z under their original names, x under the alias lib1x. Similarly, variables x and fun from module *lib2* are accessed under the indicated aliases.

11. The syntax of mode declarations

Each program variable can be declared to have a *mode*. A *mode descriptor* specifies the SETL type of a variable, and in addition may give structural information about it, e.g. its size, its relationship to other program variables (included in, subset of, based on). The user can introduce *mode names* to refer to mode descriptors and use these names somewhat as if they were SETL types. Mode descriptors can be of 4 types: a) Basic modes, b) derived modes, c) composite modes, and d) based modes.

1. Basic modes

These are the modes of atomic types. They can be qualified with a range specifier, giving minimum and maximum size (or value)

int (range)
char (range)
bits (range)
real
blank
label

The range specifier has the form n1....n2, where n1 and n2 are integers. It is always optional. The range specifier (0...n2) can be abbreviated (n2)

2. Derived modes

The user can introduce new mode names with the declarative statement:

mode modename: <mode descriptor>

modename can be used subsequently in the program as a valid mode descriptor.

3. Composite modes

The mode descriptors for sets, tuples and procedures are constructed recursively from other modes, using the following templates:

a) set {model} (size)
describes a set whose elements have mode *mode1*, and whose expected size is *size*. As elsewhere, this parameter is optional.

b) tuple <mode2> (size)
describes a homogeneous tuple whose components have mode *mode2*.

c) tuple <model, mode2, mode3 ...>
describes a tuple of known length whose components have specified modes.
For these 3 descriptors, the keywords set and tuple are optional.

d) subr (model, mode2 ...)
declares a procedure whose arguments have the specified modes

e) fnct (model, mode2 ...) moder
declares a function and the mode of the value it returns.

f) map (model) mode2
declares a map, i.e. a set of pairs. The domain of the map has mode *mode1*, and its range has mode *mode2*.

g) map(model, mode2 ...)moder
declares a multivariate map.

h) In the 2 preceding cases, the keyword smap can be used to specify a single-valued map.

4. Based modes

Based modes introduce structural relations among specific program variables. The building block of based mode declarations is the membership mode:

a) ∈ varname
where *varname* is the name of a program variable which has previously been declared to be a set. Subset declarations take the form:

b) set {∈s}
which describes a subset of set s, i.e. a set of elements of s.

A based map is declared as

c) map (s) moder, etc.

In addition subsets of a given set which can advantageously be described by membership bits or by bit-strings, are declared using the keyword subset.

For example:

d) subset(s)

The semantic restrictions which apply to these declarations will be described elsewhere.

IV. Lexical Changes

1. All SETL keywords are underlined. The compiler allows the user to select one of two keypunching conventions, either using periods after keywords or merely treating them as reserved names.

2. The macro processor uses a syntax similar to the one given in On Programming. Macro definitions have the form

```
macro <name> '(' <namelist1>; <namelist2> ')';
      <body>
end   <name>;
```

The first namelist contains the macro's arguments. The second lists names occurring within the macro which are to be replaced with unique names on each expansion. Either list may be omitted. If $namelist_2$ is omitted the semicolon is unnecessary; if both lists are omitted the parentheses may be omitted. End must be followed by the macro name and may be followed by additional tokens.

Macros may be nested to any depth; inner definitions are absorbed when the macros containing them are expanded.

Macros can contain compile-time variables called *params*.

3. Params

Params are compile time variables. They may appear anywhere in a program. They are designated as params and assigned integer values by their first appearance in a statement

param name = *expression*;
where *expression* can contain params, integer constants, parentheses and the operators +, -, /, and *.

Macros and params may occur between modules, i.e. after a module's end statement and before the start of the next following module. Macros or params occurring in this position are global to the entire compilation in which they occur. Macros and params defined at the beginning of a module are global to the module, while those defined within a routine are local to that routine.

Macro and parameter names may be 'stropped'.

4. Scanner and listing controls:
Several special cards are allowed in SETL program. They begin in column 2 and control scanner operations:

.EJECT	starts a new page of the listing
.TITLE 'string'	starts a page with title 'string'
.COPY 'file'	reads input from *file* till an end of record is encountered.

V. Operations and Systems Constants

1. Binary operators

Operator	precedence
AND	1
OR	1
IMP	2
IN	2
NOTIN	2
EQ	2
NE	2
INCS	2
SUBSET	2
GE	2

Operator	precedence	
LE	2	
GT	2	
LT	2	
WITH	3	
LESS	3	
LESSF	3	
MIN	3	
MAX	3	
REPL	3	string replication (new)
+	4	
-	4	
||	4	
//	4	
*	5	
/	5	
**	6	
IS	7	

User defined binary operators have precedence 8.

Unary Operators

All unary operators have precedence 8.

arb

not

type

dec

oct

top

bot

-

#

BUILT-IN Functions

random
pow
npow
domain
range
atom
newat
time

System Constants

NULC
NULB
NL
MULT
CM
INT
REAL
CHARS
BITS
BLANK
LABEL
SUBR
FNCT
TUPL
SET
TRUE
FALSE