SETL User Manual

David Shields

SETL Project
Department of Computer Science
New York University
Courant Institute
251 Mercer Street
New York, New York 10012

Version: 30

24 July 1984

The user manual describes the NYU LITTLE implementation of SETL as defined by THE SETL PROGRAMMING LANGUAGE by Robert B. K. Dewar, March 12, 1980.

1.0 Introduction

This document describes the NYU LITTLE implementation of the SETL Language; it is organized so that material common to all implementations is presented first, followed by material applicable to particular implementations.

The system consists of a translation phase and an execution phase. The translation phase transforms SETL source into an internal form known as Q2. The execution phase consists of an interpreter which interprets the generated Q2 instructions. The translation phase consists of three subphases, known as PRS, SEM and COD. The execution phase is known as LIB.

The system is written in an implementation language called LITTLE. The use of LITTLE is usually transparent to the SETL user, but does show up in some areas such as program parameters and input/output. Certain arbitrary limits are also imposed, though these should not be encountered in ordinary use.

This implementation supports the language as described in the SETL Language Reference Manual (March 12th, 1980), except as noted below:

1.1 Features Not Implemented

1. CASE OF variant of CASE statement

2. CASE OF variant of CASE expression

3. FROMB, FROME for string right operand

4. VAL

5. REVERSE

6. REPLACE

7. Multiple uses of names.
   A name used in a PROGRAM statement cannot be used to name a variable or other procedure. A name used in a VAR statement cannot be used as the name of a formal parameter.

8. EOF for GET
   EOF doesn't work for GET. You must test the argument, which is set to OM when the end of the file is reached:

```
       GET(afile,datum);
       IF datum=OM then ... end-processing ; end if;
```

1.2 Features Implemented Differently

1. Several input/output features implemented differently. See section 3 for details.

2. The ELSE clause in CASE expression is required.

3. Assignments of the form:

   ```
   [x,y] := z;
   ```

   where z is OM or not a tuple cause compile or run-time errors.

4. The semantics of the assert statement

   ```
   assert left_hand_side := expression;
   ```

   is defined to mean 'test whether left_hand_side equals expression; if they are unequal, print an error message and assign expression to left_hand_side.' This feature is useful during debugging of programs, in particular in connection with the run-time error limit feature described below.

1.3 Additional Features

These are described in section 4. Note also that section 4 describes procedures which are implemented, but not yet described in the language reference manual.

2.0 Standard Procedures

This section describes the standard procedures of the SETL system, which should be available in all implementations. It includes all the standard procedures, including some which are also described in the Reference Manual, in which case the description here describes how the procedures operate in this implementation.

The implementation provides several "pattern match" procedures based on those of the SNOBOL4 language. The procedures are ANY, BREAK, LEN, LPAD, MATCH, NOTANY, RANY, RBREAK, RLEN, RMATCH, RNOTANY, RPAD, RSPAN and SPAN. They have the form:

```
        proc(RW str, RD exp)
```

The first argument is a string and the second argument is either a string or integer. The procedure attempts to match part of the first operand string. If the match can be done, the procedure "succeeds" and returns the matched substring, and also removes the matched string from the argument. If the match cannot be done, the procedure "fails" and returns OM. Appendix A contains definitions of these procedures in SETL.

For historical reasons, the implementation supports several procedures which are recognized but are to be considered undefined. They will not be described here. Pending their removal, the only effect of their presence is that their names are reserved words, as follows:

```
   GETK  PUTK  REWIND
```

ANY(RW str, str)

ANY is a pattern match which succeeds if the first character of the first argument occurs in the second operand. ANY fails if the first argument is null.

BREAK(RW str, str)

BREAK is a pattern match which succeeds if the first argument contains a character which is in the second argument. BREAK matches the initial substring of the first argument which does not contain any characters which are in the second argument.

CLOSE(file)

CLOSE terminates the input/output to a file initiated by a prior call to OPEN.

EJECT() or EJECT(filename)

EJECT causes a page eject on the specified file. If no argument is specified, the eject occurs on the standard output file. If the specified file is not the standard output file, it must have been opened with 'PRINT' specified as second argument to OPEN.

EOF

EOF returns TRUE if the last input operation encountered end of data; otherwise EOF returns FALSE.

GET(file, WR lhs, ...)

GET reads successive lines from the specified file, and assigns them as strings to the corresponding left hand sides. Any input values which were not available because of encountering the end of file cause the corresponding arguments to be set to undefined (OM).

GETB(file, WR lhs, ...)

GETB reads binary values from the file specified by the first argument. GETB should be used only for files created using the PUTB binary output procedure. Any input values which were not available because of encountering the end of file cause the corresponding arguments to be set to undefined (OM).

GETEM(WR lhs, WR lhs)

GETEM assigns the current run-time error mode to its first argument and assigns the current run-time error limit to its second argument. See section 5 for further information.

GETIPP(str)

GETIPP obtains a program parameter value specified by its argument, which has the form

```
'NAME=DEFVAL/ALTVAL'
```

GETIPP makes available to the user the same scheme used to obtain program parameters that is used by the SETL system, as described in section 6. GETIPP returns an integer value.

GETSPP(str)

GETSPP is similar to GETIPP except that it returns a string.

HOST() or HOST(...)

The usual, and always acceptable, implementation is for HOST to return the undefined value (OM). Particular implementations may provide other features; consult section 8 for details.

LEN(RW str, int)

LEN is a pattern match procedure. It succeeds if the first argument length is as large as the second argument. An error occurs if the second argument is negative.

LPAD(str, int)

LPAD returns the result of padding the first argument to the length given by the second argument. Blanks are added, as needed, to the left. If the length of the first argument exceeds the value of the second argument, the first argument is returned unchanged.

MATCH(RW str, str)

MATCH is a pattern match which succeeds if the second argument occurs as an initial substring of the first argument.

NOTANY(RW str, str)

NOTANY is a pattern match that succeeds if both arguments are not null, and the first argument begins with a character not contained in the second argument.

OPEN(file, mode)

OPEN opens a file specified by the first argument. The acceptable forms of the first argument are machine-dependent in that the value is a file name as defined by the execution environment. The second argument is a string, and must be one of the following:

'BINARY-IN'	binary input
'BINARY'	same as 'BINARY-IN'
'BINARY-OUT'	binary output
'TEXT-IN'	text input
'TEXT'	same as 'TEXT-IN'
'TEXT-OUT'	text output
'CODED'	same as 'TEXT'
'CODED-IN'	same as 'TEXT-IN'
'CODED-OUT'	same as 'TEXT-OUT'
'PRINT'	text file for printing

The second argument specifies the format of the file and also whether the file is to be read or written. In mixed-case implementations, the second argument can be written using all upper-case or all lower-case characters; for example, 'TEXT' or 'text', but not 'Text'. Open may be used as a logical function. It returns TRUE if it was able to open the file, and FALSE if it was not. Attempting to use the file in an illegal manner will produce a run time error.

PRINT(expr, ...)

PRINT converts its arguments into strings as appropriate and writes them to the standard output file.

PRINTA(file, expr, ...)

PRINTA is similar to PRINT, except that output is sent to the file specified by the first argument.

PUT(file, expr, ...)

PUT writes text lines to the file specified by the first argument. Each expression argument must be a string and results in a single line being written to the specified file.

PUTB(file, expr, ...)

PUTB writes binary values to the file specified the first argument. Each expression is written in an appropriate internal form which can later be read in using the GETB procedure.

RANY(RW str, str)

RANY is similar to ANY except that it matches from the right.

RBREAK(RW str, str)

RBREAK is similar to BREAK except that it matches from the right.

READ(WR lhs, ...)

READ reads values from the standard input file. Any input values which were not available because of encountering the end of file cause the corresponding arguments to be set to undefined (OM).

READA(file, WR lhs, ...)

READA is similar to READ except the first argument specifies the file which is to be read.

RLEN(RW str, len)

RLEN is similar to LEN except that it matches from the right.

RMATCH(RW str, str)

RMATCH is similar to MATCH except that it matches from the right.

RNOTANY(RW str, str)

RNOTANY is similar to NOTANY except that it matches from the right.

RPAD(str, int)

RPAD is similar to LPAD except that any needed blanks are added to the right of the first argument value.

RSPAN(RW str, str)

RSPAN is similar to SPAN except that it matches from the right.

SETEM(integer, integer)

SETEM sets the values of the run-time error mode and error limit according to the values of its first and second arguments, respectively. See section 5 for further information.

SPAN(str, str)

SPAN is a pattern match which matches the longest initial substring of its first argument which consists solely of characters in the second argument. SPAN must match at least one character if it succeeds.

TITLE() or TITLE(str)

TITLE with no arguments disables generation of titles. TITLE with an argument enables titling and establishes the argument as the title string of the next page. The implementation differs from the definition in that TITLE does not cause a page eject but leaves the standard output file on the last line of the current page so that the next print statement causes a page eject. Hence several calls to TITLE, without intervening actions that create output, just establish the argument of the last call, if it is not null, as the title of the next page.

3.0 Input/output

The implementation of the input/output features differs in a number of ways from the definition.

If READ or READA are used to input a real value with an exponent, the letter "E" must be specified in upper case.

4.0 Additional Features

The implementation includes several additional features.

These include listing control, remote text inclusion, a macro processor, and error values. Error values are described in section 5.

4.1 Listing Control Commands

Listing control commands are used to alter the form of the source program listing. They have no other effect on compilation or execution. They always occur on a separate line which begins with the characters " .". The listing control commands are as follows:

.LIST
: Causes listing of following lines.

.NOLIST
: Disables listing of following lines.

.EJECT
: Causes generation of a new page on the standard output file.

.TITLE
: Gives a title which will appear at the top of subsequent pages. The AT program parameter may be used to request "automatic titling" which causes each new procedure definition to force an eject with a title derived from the line containing the procedure definition header. The title text cannot contain the delimiting apostrophe character.

4.2 Remote Text Inclusion

Remote text can be merged with the standard input file during compilation using the .COPY command (recall that the command begins with ' .' in the first two columns.). The command line has the form

```
 .COPY 'name'
```

and causes the specified remote text to be effectively substituted in place of the .COPY command. The remote text must be in an inclusion library. Sections of an inclusion library are specified by a line which begins with ' .=MEMBER name". A section is terminated by the next following MEMBER line or the end of the library.

The remote text may contain .COPY commands. However, the depth of nested copies is limited to five.

The ILIB program parameter specifies the file containing the inclusion library.

4.3 Macro Processor

The compiler provides a macro processor which supports macros with parameters and generated local symbols.

A macro definition has one of the following forms:

```
        MACRO macro_name; macro_body ENDM;
        MACRO macro_name (arglist);  macro_body ENDM;
```

The keyword ENDM may be followed by the macro name to increase readability. In both these forms macro_name is the macro name and macro_body is any sequence of text not containing macro_name. The argument list consists of a list of names of formal parameters and generated parameters:

```
        (fp1, fp2...; gp1, gp2...)
```

At least one formal or generated parameter must be present. The semicolon appears only if there is at least one generated parameter.

A macro with no arguments is invoked by writing its name, which is then replaced with the macro body. A macro with arguments is invoked in the form

```
        macro_name(ap1, ap2...)
```

where the number of actual parameters is the same as the number of formal parameters. Each actual parameter is any sequence of tokens not containing a comma which is not included within parentheses. An invocation is replaced by the macro body, with each instance of a formal parameter replaced by the corresponding actual parameter.

Macros with generated arguments are expanded similarly, except that a unique name is generated for each generated parameter, and each instance of the generated parameter is replaced by the corresponding name.

The text of a macro with arguments may contain embedded macro definitions; these definitions become active when the macro is invoked. Macro expansion is recursive, and outside-in.

Macros are undefined using the form:

```
        DROP macro_name1, macro_name2, ... macro_namen;
```

which undefines each of the specified names.

4.4 Synonyms

Several synonyms are recognized: OPERATOR for OP, PROCEDURE for PROC, and WHERE for ST.

4.5 CONST Name

The form "CONST name;" where name has not been previously declared is taken to be:

```
        CONST name='name';
```

i.e., a constant string.

4.6 FORALL Permitted For Iterators

To provide compatibility with previous versions, FORALL may be used to begin iterators:

```
        (FOR x IN S)... END;
        (FORALL x IN s)...END;
```

are equivalent.

4.7 ARB In CASE Tag

A CASE tag may have the form ARB S, where S is a constant set.

4.8 Backtracking

Backtracking is a powerful means of expressing search algorithms. The backtracking primitives are the OK operator, the LEV operator, the FAIL statement and the SUCCEED statement.

OK is a niladic operator which saves the current values of all variables declared with the BACK attribute. The current position within the program is also saved.

The FAIL statement has the form:

```
        FAIL;
```

The values of all variables saved by the last OK are restored. The value of the last OK is set to FALSE and execution continues.

LEV is a niladic operator which returns the number of environments currently being saved.

The SUCCEED statement has the form:

```
        SUCCEED;
```

If a program performs a series of OK's and never reaches a FAIL then the saved values will tend to overflow memory. The SUCCEED statement releases the memory used for the last OK. It asserts that the program will never try to fail out of the OK.

4.8.1 Demonstration Program Using Backtracking

```
$ The 8 queens problem : E. Schonberg, NYU

PROGRAM queens;

VAR numqueens, boardsize, board;

print('number of queens : '); read(numqueens);
print('board size : ');       read(boardsize);

board := { [i,j] : i IN [1..boardsize], j IN [1..boardsize] };
queens_backt();                 $ backtracking version .
STOP;

PROC queens_backt;

VAR used , possible  : BACK;

used := {};

(WHILE #used < numqueens )
        possible := safe(used);
        IF  (EXISTS pos IN possible ST OK ) THEN
                used WITH:= pos;
        ELSE FAIL;
        END IF;
END WHILE;

printboard(used);

END PROC;

PROC safe(used);

$ this utility procedure produces the board positions that are
$ not under attack, after the positions in -used-
$ have been occupied.

RETURN
    {pos IN board ST (FORALL u IN used ST NOT attack(pos,u)) };
END;
```

```
PROC attack(p1, p2);

$ this utility checks whether positions p1 and p2 are
$ mutually threatening.

RETURN
  (p1(1) = p2(1) )    OR                    $ same row.
  (p1(2) = p2(2) )    OR                    $ same column .
  ( (p1(1) - p1(2)) = (p2(1) - p2(2))) OR $ same up diagonal.
  ( (p1(1) + p1(2)) = (p2(1) + p2(2)));  $ same down diagonal.
END;

PROC printboard(used);

top := boardsize * '__' + '_';
row := boardsize * 'I_' + 'I';

print; print;
print(top);

(FORALL rownum IN [ 1..boardsize] )
        nextrow := row;
        (FORALL colpos IN used{rownum})
                nextrow(2*colpos) := '*';
        END;
        print(nextrow);
END FORALL;

print; print;

END PROC;

END PROGRAM;
```

4.9 Representation Sublanguage

Representation declarations are used to indicate the representation of variables, procedures, operators, and symbolic constants.

Each series of declarations can be followed by a REPR clause which has the format:

```
REPR
    <repr clauses>
END [REPR];
```

4.9.1 Location Of Representation Declarations

Representations for all variables and constants declared in a directory must occur after all global declarations and interface statements within the directory.

Representations for all variables and constants declared at the beginning of a library, program or module must occur immediately after all global-to-library, global-to-program, or global-to-module declarations and the optional interface statements within the same library, program or module.

Representations for library procedures must occur at the start of the library after the global variable declarations.

Procedures exported by programs and modules must have their representations declared in the directory. Procedures which are local to a library, program or module are declared in the library, program or module.

Formal parameters are represented implicitly by giving the representation of their procedure. These representation declarations can be duplicated within the procedure itself. Such duplicate representation declarations are checked for consistency.

4.9.2 Restrictions

Library procedures cannot have based arguments or return a based value.

Variables declared to have LOCAL basing on a base B must have their representations declared in the same scope as B. Formal parameters can only be based on bases global to module or global to directory.

Formal parameters can have LOCAL basing if their procedures are not recursive. Local variables can have LOCAL basing if their procedures are not recursive and if the variables are not backtracked.

4.9.3 REPR Statement

The REPR declaration statement consists of a series of clauses. Each clause is a BASE clause, a PLEX BASE clause, a MODE clause, or a representation clause.

The BASE clause declares a series of names to be bases. Its format is:

```
        BASE  <base name list> : <mode>;
```

A base name is either a name or the name of a constant set which has already appeared in a CONST statement.

The PLEX BASE clause declares a series of names to be plex bases. It is similar to the base clause, but begins with the word PLEX, i.e.

```
        PLEX BASE <base name list>;
```

The MODE clause creates a new name for a mode. Its form is:

```
        MODE <name> : <mode> ;
```

A representation clause gives the types of variables, symbolic constants, procedures and user defined operators, and has the form:

```
        <name list> : <mode>;
```

4.9.4 Modes

This section gives an informal description of the productions for <mode> along with their meanings.

general	variable can have any type
*	synonym for general
ELMT <name>	element of base
<name>	mode declared in prior MODE clause
INTEGER	integer
INTEGER cexp ... cexp	integer in range cexp...cexp
REAL	
STRING	
ATOM	
ERROR	
UNTYPED REAL	
UNTYPED INTEGER	
SET(mode)	
SET	set(*)
TUPLE(mode)	tuple of mode
TUPLE(mode) (n)	tuple of mode, estimated length n
TUPLE	tuple(*)
TUPLE(m1, ..., mn)	mixed tuple : component i has mode mi

```
MAP(m1, ..., mn) mn+1   (ambiguous) map
SMAP(m1, ..., mn) mn+1  single valued map
MMAP{m1, ..., mn} mn+1  map likely to be multivalued
MAP                     synonym for MAP(*) *
SMAP                    synonym for SMAP(*) *
MMAP                    synonym for MMAP{*} *
MAP(m1, ..., mn)        synonym for MAP(m1, ..., mn) *
SMAP(m1, ..., mn)       synonym for SMAP(m1, ..., mn) *
MMAP(m1, ..., mn)       synonym for MMAP{m1, ..., mn} set(*)
MMAP(m1, ..., mn) mn+1  synonym for MMAP{m1, ..., mn} set(mn+1)
LOCAL <set or map mode>
REMOTE <set or map mode>
SPARSE <set or map mode>
PACKED <tuple or map mode>

PROC(m1, ..., mn) mn+1  procedure
PROC(m1, ..., mn)
PROC
PROC() m

OP(m1, ..., mn) mn+1    Operator
OP(m1, ..., mn)
OP
```

4.10 Structuring Large Programs

This section uses the syntax conventions of chapter 8 of the reference manual to describe the means provided for organizing large SETL programs.

A SETL program may consist of a directory, a program unit and also module and library units.

```
        module_prog     {library_unit}
                        direc_unit prog_unit
                        {module_unit}
                        {decl_repr}

        direc_unit      DIRECTORY tok_nam ;
                          {decl}
                          PROGRAM tok_nam - tok_nam : dir_spec
                          {MODULE tok_nam - tok_nam : dir_spec }
                        END [DIRECTORY [...] ] ;

        decl            VAR tok_nam {, tok_nam} [: BACK] ; |
                        CONST declcon {, declcon} ; |
                        INIT tok_nam := constant
                          {, tok_nam := constant} ; |

        declcon         toknam [ = constant ]

        decl_repr       REPR repr {repr} END [REPR [...]] ;
```

The DIRECTORY unit gives the relationships between the other units and includes the declarations of any global variables and constants. The PROGRAM unit contains the main program. The MODULE and LIBRARY units contain groups of related procedures.
Units referenced within the directory have names of the form:

```
        dir_name  - unit_name
```

dir_name is the name given in the DIRECTORY statement, and unit_name is the name of the unit.

4.10.1 Directory Specification

A directory specification has the form;

```
        dir_spec        {dir_item} {decl_repr}

        dir_item        reads_item | writes_item |
                        exports_item | imports_item

        reads_item      READS tok_nam {, tok_nam} ; |
                        READS ALL;

        writes_item     WRITES tok_nam {, tok_nam} ; |
                        WRITES ALL;

        imports_item    IMPORTS pspec {, pspec} ;

        exports_item    EXPORTS pspec {, pspec} ;

        lib_item        LIBRARIES tok_nam {, tok_nam} ;

        pspec           tok_nam [ ( pspeca {, pspeca} ) ]  |
                        tok_nam [ ( {pspeca ,} pspeca ( * ) ]

        pspeca          RD [tok_nam]
                        WR [tok_nam]
                        RW [tok_nam]
                        [tok_nam]
```

A reads_item lists the global variables and constants which may be read within a unit. A writes_item lists the global variables to which new values may be assigned within a unit.

An imports_item lists the procedures defined elsewhere which are used within a unit. An exports_item lists the procedures defined within the unit which can be imported by other units.

A lib_item lists the libraries used within a unit and causes all of the procedures of the library to be imported.

4.10.2 PROGRAM Unit

The program unit defines the main program:

```
        prog_unit       PROGRAM tok_nam - tok_nam ;
                          {lib_item}
                          dir_spec
                          {decl decl_repr}
                          {stmt}
                          {refine}
                          {routine}
                        END [PROGRAM [...] ] ;
```

The program unit is similar to the program part of a short SETL program which consists solely of a program unit and procedures.

4.10.3 MODULE Unit

A module_unit contains a set of related procedures:

```
        module_unit     MODULE tok_nam - tok_nam ;
                          {lib_item}
                          dir_spec
                          {decl decl_repr}
                          routine
                          {routine}
                        END [MODULE [...] ] ;
```

The procedures may access global variables declared in the directory. Directory specifications within the MODULE may be included for documentary purposes. If given, they must agree with the specifications in the DIRECTORY.

4.10.4 LIBRARY Unit

A library unit has the form:

```
        library_unit    LIBRARY tok_nam ;
                          {lib_item}
                          {exports_item}
                          {decl decl_repr}
                          routine
                          {routine}
                        END [LIBRARY [...] ] ;
```

A library_unit is similar to a module_unit except that it may not access global variables.

5.0 Compilation And Execution Errors

During the course of translation or execution the system may detect an error.

5.1 Translation Errors

Translation errors cause generation of an error message on the standard output file. Since the compiler runs in three phases, three distinct listing files may be produced. Most ordinary syntactic errors will be caught by the first (PRS) phase. Errors uncovered by the SEM and COD phases are usually more global in nature. The error recovery scheme used is quite weak, and error messages after the first few may not be too reliable.

Several program parameters are related to error processing. The PEL, SEL and CEL parameters respectively specify the maximum number of errors permitted during the PRS, SEM and COD phases. A choice of low values can force compilation to terminate after the detection of a small number of errors. The UV option can be used to identify undeclared variables.

The compiler contains a number of tables which in some cases may overflow. Overflow is reported by the generation of an error message, usually containing the internal name of the table, and compilation is abnormally terminated. Errors of this sort require that the program be made "smaller", for example, by dividing a large procedure into several smaller procedures.

5.2 Execution Errors

Processing of errors during execution (run-time) is controlled by the REL and REM program parameters, and by the GETEM and SETEM builtin procedures. The system deals with errors according to the error mode in effect. Certain errors (deemed "fatal") always cause execution to abnormally terminate. Otherwise error handling depends on current error mode and error limit, as follows:

1. EM=1

   Ignore error if possible, yield OM as value.

2. EM=2

   Print error message, increment error count. If the cumulative error count exceeds the error limit, abnormally terminate execution. Otherwise, yield OM as value and continue execution.

3. EM=3

   Increment error count, continue execution. This mode is used by the SETL optimizer; its use is not suggested.

4. EM=4

   Print error message, increment error count. If the cumulative error count exceeds the error limit, abnormally terminate execution. Otherwise, return a special "error value" and continue execution. An error value is a special internal object which contains an encoding of the point in the program at which the error occurred. Error values may be printed, in which case the description of the program point is converted into readable form.

The initial error mode is specified by the REM program parameter. The initial error limit is specified by the REL program parameter. The procedure GETEM(a, b) assigns the current error mode to a and the current error limit to b. The procedure SETEM(a, b) sets the error mode to a and the error limit to b.

5.3 TRACE Statement

The current implementation recognizes but does not fully process a trace statement which has the form:

```
        trace_stmt      trace_key trace_opt {,trace_opt}
        trace_key       TRACE | NOTRACE
        trace_opt       CALLS | STATEMENTS | name
```

This feature is not fully working. The CALLS and STATEMENTS traces are available. TRACE name is available in experimental form on some but not all systems.

5.4 DEBUG Statement (System Checkout)

The current implementation includes a debugging statement used for system checkout which has the form:

```
 DEBUG dopt {, dopt};
```

It is not needed for ordinary use; it is described here for completeness. Users of an experimental bent can use it to probe system innards; also on occasion requests may be made to use this feature to assist in remote diagnosis of a problem. Not all options need to work on every system.

5.4.1 PRS Options

The options that apply to PRS are:

PTRM0	disable macro-processor trace
PTRM1	enable macro-processor trace
PTRP0	disable parser trace
PTRP1	enable parser trace
PTRT0	disable token trace
PTRT1	enable token trace
PRSOD	list "open" tokens
PRSPD	list polish and x-polish tables
PRSSD	list symbol table

5.4.2 SEM Options

The options that apply to SEM are:

STRE0	disable entry trace
STRE1	enable entry trace.
STRS0	disable ASTACK trace. ASTACK is the argument stack.
STRS1	enable ASTACK trace
SQ1CD	list Q1 code
SQ1SD	list Q1 symbol table
SCSTD	list CSTACK.

5.4.3 COD Options

The options that apply to COD are:

CQ1CD	list Q1 code
CQ1SD	list Q1 symbol table
CQ2CD	list Q2 code

5.4.4 LIB Options

The options that apply to LIB are:

RTRE0	disable entry trace of SETL LIB procedures
RTRE1	enable entry trace to procedures in SETL LIB
RTRC0	disable code trace
RTRC1	enable code trace
RTRG0	disable garbage collector trace
RTRG1	enable garbage collector trace
RGCD0	disable dynamic storage dumps during garbage collection
RGCD1	enable dynamic storage dumps during garbage collection

RDUMP	dump dynamic storage to file specified by DUMP= program parameter for later examination using DMP program
RGARB	force a garbage collection

5.5 Expiration Check

The distributed system may contain an expiration check. During the month before expiration, a warning message of the form

```
        nnn DAYS TO EXPIRATION.
```

is generated at the start of the standard output file. Consult the system manager if this message appears, so arrangements to acquire a new copy of the SETL system can be made before expiration. Expiration is signaled by issuing the message

```
        EXPIRED, OBTAIN NEW COPY.
```

and then execution is abnormally terminated.

6.0 Program Parameters

This section describes the program parameters supported by the compiler and interpreter. These parameters are specified as part of the command line used to invoke the SETL system. The system program parameters are described in the LITTLE format

```
        NAME=DEFVAL/ALTVAL
```

where NAME is the parameter value, DEFVAL is the default value if the parameter is not otherwise specified, and ALTVAL is the value taken if the parameter name alone is given. Parameter values are either decimal integers or character strings. For example, given

```
        P=0/1
```

then if P not mentioned, value 0 is implied. If P alone specified, then value 1 is implied. If P=n specified, the value n is implied.

A number of parameters have the form NAME=0/1. Such values are logical switches in that they select one of two cases, according as value is zero or non-zero. In the latter case, the option is said to be ENABLED or SELECTED.

Each parameter description mentions the phases for which the parameter has meaning. Note that the parameter codes have been chosen so that the same list can be passed to all phases; i.e., the same parameter does not have differing meanings in different phases.

Parameter values are sought left to right so that, for example,

```
        P=1,LIST,P=2
```

yields value 1 for parameter P.

Parameters specifying files tend to be machine-dependent; see section 8 for the conventions and defaults used for a particular implementation.

ASM=0/1 (COD)

Controls whether assembly code output for hard code is produced.

ASSERT=1/2 (LIB)

Specifies processing of ASSERT statements, as follows:

- 0 - Treat all assertions as no-ops; the expression of the assert statement is not evaluated.
- 1 - Test all assertions, yield error on assertion failure.
- 2 - Test all assertions, print true assertions, yield error on assertion failure.

AT=0/1 (PRS)

Controls whether automatic titling is in effect. This option is enabled by AT=1. If enabled, then each new procedure causes a new page on the listing. Note that a listing is produced only if the listing option (LIST=1) is specified.

BACK=0/1 (COD)

Controls whether code supporting backtracking is generated. This option must be selected if the program uses backtracking.

BIND=0/filename (SEM)

Specifies BINDer file used to merge results of prior compilations. This feature is intended to permit separate compilation. It is under development and will not be described further.

CA=0/0 (COD)

Gives length of constants area. If zero specified, value used is half that given by H= parameter. A value less than 1024 is multiplied by 1024; for example CA=2 is equivalent to CA=2048.

CEL=1000/1000 (COD)

Specifies the COD error limit. If more than the specified number of errors are detected by the COD phase, compilation is terminated.

CSET=EXT/POR (PRS)

Specifies character set of SETL source, as follows:

```
 POR    Portable character set
        <<  for left set bracket
        >>  for right set bracket
        (/  for left tuple bracket
        /)  for right tuple bracket

 EXT    Extended set, consisting of portable set plus
        {   for left set bracket
        }   for right set brace
        [   for left tuple bracket
        ]   for right tuple bracket
        |   for ST
        !   for ST
```

The above characters are the standard ASCII extended set. The actual character available is of course machine dependent.

CTRACE=0/1 (LIB)

Controls whether the procedure call trace feature of the SETL Debugger is activated. This feature is under development.

DEBUG=0/1 (LIB)

Controls whether the SETL Debugger is activated. This feature is under development.

DITER=0/1 (SEM)

Controls whether compiler may assume that within loops objects being iterated over are not modified. The default value is consistent with the definition in that iterations always cause a copy to be made of the object being iterated over, thus permitting modification of the object. When selected, the compiler will not cause the copy to be made.

DUMP=filename (LIB)

Specifies file to receive internal dump of SETL heap; this feature used for system checkout. This feature also permits programs with a lengthy initialization process to be run with the initialization done only once. For example, consider the program

```
program main;
initialize();
debug rgarb; debug rdump;
mainbody();
...
```

Then if the program is run and the DUMP file is specified, further runs can be done by using the DUMP file generated as the Q2 file.

ETIM=0/1 (all)

Specifies whether elapsed execution time should be reported on the standard listing file.

H=0/0 (COD,LIB)

Specifies initial heap length. A value less than 1024 is multiplied by 1024; for example, H=2 is equivalent to H=2048. This value need be specified only to set an initial large heap size in order to prevent system expanding heap in steps when prior runs have revealed that a large heap is needed. The default of zero specifies that the SETL system is to set up a heap of reasonable size that should be sufficient for most applications.

I=filename (LTL)

Specifies the standard input file.

ILIB=filename (LTL)

Specifies the file containing the inclusion library. This parameter has meaning only if the .COPY command is used to include remote text.

LCP=0/1 (all)

Controls whether program parameters are listed on the standard listing file.

LCS=1/0 (all)

Controls whether execution statistics are listed on the standard listing file.

L=filename (LTL)

Specifies name of standard output (listing) file.

LIST=0/1 (PRS)

Controls whether listing of source program is produced. LIST=1 must be selected to produce a listing.

MAX_HEAP=0/ (LIB)

Allow heap expansion to specified number of SETL words. Zero, the default, is interpreted as 'take what is left in the process' address space minus what is expected to be needed for i/o buffers' (cf. NOF program parameter).

MEAS=0/1 (PRS)

Controls whether program measurements are to be made of time spent in LIB phases. This feature is under development and is currently not of interest to the ordinary user.

MLEN=1000/ (PRS)

Gives maximum number of tokens allowed in single macro body.

NOF=5/ (LIB)

Number Of Files that are open simultaneously. This parameter assures that the dynamic storage management routines leave enough address space to the operating system so that the user can have NOF open files.

OPT=0/1 (PRS, SEM, OPT, COD)

Controls whether global optimization is in effect. This option has effect only if the SETL optimizer is available.

PEL=1000/1000 (PRS)

Gives the PRS error limit. If more than the specified number of errors are detected during the PRS phase, compilation is abnormally terminated.

PFCC=1/0 (LTL)

Controls whether standard output file contains carriage control information. If PFCC=0 selected, then no carriage control is provided.

PFCL=0/80 (LTL)

Specifies characters per line of standard output file, including the column used for carriage control. PCFL=0 specifies the default line length. PFCL=80 useful when output is directed to terminal.

PFLL=0/0 (LTL)

Controls, in conjunction with PFPL parameter, whether a line limit control is to be enforced on the standard output file (see description of PFPL parameter).

PFLP=60/0 (LTL)

Specifies number of lines per page on the standard output file.

PFPL=100/0 (LTL)

Specifies print file page limit for standard output file. This parameter, in conjunction with PFLL parameter, controls whether a page limit control is to be enforced on the standard output file. There are several cases, as follows:

```
 PFPL=0,PFLL=0
        no limits enforced
 PFPL=n,PFLL=0   (n>0)
        limit of n pages or n*PFLP lines enforced
 PFPL=0,PFLL=n  (n>0)
        limit of n lines enforced
 PFPL=n,PFLL=m  (n>0,m>0)
        limit of n pages or m lines enforced
```

The default may be set to PFPL=0,PFLL=0 for a particular site if it is not customary to enforce print file limits.

POL=filename (PRS, SEM)

Specifies the intermediate "Polish" file created by PRS and read by SEM.

Q1=filename (SEM,COD)

Specifies the intermediate Q1 file created by SEM and read by COD.

Q2=filename (COD,LIB)

Specifies the Q2 file created by COD and read by LIB.

REL=0/0 (LIB)

Specifies the Run-time Error Limit. When more than specified number of errors are detected, execution is abnormally terminated. See also the section on "errors".

REM=2/2 (COD)

Specifies the run-time error mode. The need for a value other than the default should not arise in usual practice. See section on "errors" for more information.

REPRS=1/1 (SEM, COD)

Controls effect of declared representations on code generation. REPRS=0 causes compiler to ignore any representation declarations as much as possible, though declarations are checked for consistency. Specify REPRS=2 to give same result as REPRS=1 with additional effect of printing a message during code generation for each conversion generated between non-primitive values; this requires use of LIST qualifier to have any effect.

SB={}/<<>> (LIB)

Specifies characters for printing set brackets.

SEL=1000/1000 (SEM)

SEM Error Limit which, if exceeded, causes SEM to abnormally terminate.

SEQ=LS (PRS)

Selects whether line numbers (SEQ=L), statement numbers (SEQ=S), both (SEQ=LS), or neither (SEQ=), are listed in source listing.

SIF=0/1 (SEM, COD)

Controls whether intermediate files are to be saved. The default is to drop intermediate files. However, SIF=1 must be specified to save the Q1 file when the binder is used.

SNAP=0/1 (LIB)

Controls whether SNAP dump of recent values of variables is to be produced when run-time error detected. Specify zero value to suppress this dump.

SOCASE=0/0 (LIB)

Controls the case in which values of string-valued primitives are returned. SOCASE=0 specifies the default case; SOCASE=1 specifies lower case; SOCASE=2 specifies upper case. This option is useful in porting programs between two systems which have different default cases; for example, VAX VMS implementation has upper case as default while VAX UNIX implementation has lower case as default. Currently this option affects only the case of values returned by TYPE operator. In general, TYPE can be avoided by using

```
IS_INTEGER x
```

instead of

```
TYPE x = 'INTEGER'
```

SQ1=filename (SEM, OPT, COD)

Specifies the file containing SQ1 form of program text. This is used by the SETL optimizer, which is under development, and will not be needed for normal use.

SSM=filename (PRS, OPT)

Specifies the file to contain source listing in form of SETL map. This file is used by the SETL optimizer and also by some of the SETL measurement facilities.

ST=0/0 (COD)

Specifies the initial length of run-time symbol table. If ST=0 is specified, the system makes a reasonable guess; otherwise the specified number of words are allocated for the run-time symbol table. To simplify writing of large values, specification of ST=n where n<1024 is taken as though ST=n*1024 had been written.

STLO=1/1 (COD)

Specifies initial statement number for measurement. Used in conjunction with MEAS and STHI parameters.

STHI=0/0 (COD)

Specifies final statement number for measurement. Used in conjunction with MEAS and STLO parameters.

STMT=1/2 (COD)

Controls generation of Q2 STMT quadruples used to record program position during execution. STMT=0 causes no STMT quadruples to be generated, so that error messages will indicate first statement in the procedure in which the error occurred. STMT=1 causes a STMT quadruple to be generated for each statement. STMT=2 causes an additional STMT quadruple to be generated after each procedure call; this is only needed for more accuracy in program profiles generated using the various SETL measurement facilities.

STRACE=0/1 (LIB)

Controls whether dynamic trace giving statement number of each statement executed is produced. The statement numbers are given in the PRS listing (which is generated only if the LIST option selected).

TB=[]/() (LIB)

Specifies the characters for "tuple brackets" to be used when printing values.

TERM=filename (LTL)

Identifies the file to receive "terminal" output. If requested, this file will contain copies of error messages, warnings, etc., written to standard output file (see L= parameter). Note that implementations for interactive systems will usually produce a terminal file which will be sent to the user's terminal. Other implementation will produce the output on a separate file. The option TERM=0 always suppresses the terminal output.

TERMH=0/1 (LTL)

Controls whether a terminal "header line" is to be sent to the terminal when a program starts up.

TERMP=implementation-dependent (LTL)

Specifies the prompt character for terminal input. TERMP=0 gives no prompt. The default is to prompt on terminal input; the prompt character varies from implementation to implementation.

TITLE=0/1 (LIB)

Controls whether LIB will produce standard output file which is divided into pages and has title for each page.

TP=0/0 (PRS)

Controls whether PRS is to terminate compilation if any errors detected.

UPD=0/1 (PRS)

Controls whether source file is assumed to be in LITTLE UPD library format (UPD=1), permitting direct compilation of such library files.

UR=0/1 (SEM)

Controls whether SEM will give warning message for variables for which no representation declaration has been given. This option has meaning only if REPRS=1 is in effect.

UV=0/1 (SEM)

Controls whether SEM will give warning message for undeclared variables.

XPOL=filename (PRS,SEM)

Specifies the intermediate "X-Polish" file created by PRS and read by SEM.

6.1 Debugging Parameters

The system recognizes a number of program parameters which are used for system checkout and should not be needed in normal use. Note however, that the system will search for these parameters in the option string so that, for example, GDUMP can not be used as name of user parameter for GETIPP or GETSPP.

CHK	SEM	check code fragments
CQ1CD	COD	list Q1 code
CQ1SD	COD	list Q1 symbol table
CQ2CD	COD	list Q2 code
CQ2SD	COD	list Q2 symbol table
CTRE	COD	trace entry to procedures
DUMP	LIB	names file to receive storage dump
ENTRY	LIB	procedure entry trace
ET	all	obtain dumps after errors
GDUMP	LIB	save dynamic storage image on file
GTRACE	LIB	trace garbage collector
IDUMP	LIB	generate initial dynamic storage dump
MT	PRS	trace macro processor
PD	PRS	dump polish text
PT	PRS	parser trace
SD	PRS	symbol table dump

SQ1CD	SEM	list Q1 code
SQ1SD	SEM	list Q1 symbol table
STORES	LIB	enable stores trace
TRE	SEM	trace procedure entry
TRP	SEM	trace polish file
TRS	SEM	trace astack
TT	PRS	trace token stream

7.0 Interface To Procedures Not Written In SETL

The interface described in this section is currently available only for the VAX/VMS implementation and the description is for this implementation. It is planned to make the interface available for other SETL implementations, at which time the documentation will be suitably revised.

The SETL programmer can access procedures written in other languages, such as FORTRAN or MACRO, thus making it possible to use system features not available in the standard SETL system and to install high-efficiency options.

On the VAX it is particularly easy to call procedures written in a variety of other languages, since VAX VMS uses a standard system-wide calling sequence. This description will emphasize use of FORTRAN as the language used to construct a SETL-to-external facilities interface. However, the same methods can be readily adapted for use with other languages.

The system works as follows. A FORTRAN 'driver' program passes to SETL an array giving the addresses of the FORTRAN procedures which are to be made available. Communication between SETL and FORTRAN involves three new SETL primitives: PUTF, CALLF and GETF. PUTF stores SETL values in a communication area. CALLF passes control to a FORTRAN procedure whose argument list is formed from a specified slice of the communication area. GETF is then used to retrieve values from the communication area and make them available to SETL. The types of values that can be communicated are SETL integers, reals, strings, tuples of integers or tuples of reals.

The variant SETL LIB, supplemented with the user-provided procedures, can then be run by using the standard SETLX command. All this permits the user to provide his own extension to SETL.

To use the interface, the user must supply the main program which identifies the external procedures to be called using PUTF. This is done by passing an array whose components are the addresses of the procedures to be invoked; the I-th entry of this array must be the address of the procedure to be called when PUTF is called with a first argument having value I.

For example, the following shows the FORTRAN main program text needed to define procedures SUB1 and SUB2 as the first and second procedures, respectively, to be accessed using PUTF:

```
        EXTERNAL SUB1,SUB2
        INTEGER EARA(2)
        EARA(1) = %LOC(SUB1) ! ADDRESS OF FIRST PROCEDURE
        EARA(2) = %LOC(SUB2) ! ADDRESS OF SECOND PROCEDURE
        CALL STLINT(EARA,2) ! PASS TWO EXTENSION PROCEDURE
                         ! TO SETL
        END
```

7.1 Representation Of Values, Datatypes Supported

The interface uses data representations which reflect the basic hardware representation of numeric and string quantities. SETL values are put into this form when presented to interface procedures; results returned from interface procedures initially use the same form, but are then converted into the internal form used by the SETL system.

The interface supports the following SETL data types: integer, real, string, tuple of integers, or tuple of reals.

1. Scalar Integer SETL integers are represented as VAX longwords; the value spans the full hardware range except for the special 'undefined integer' value used by SETL (which has hex value '80000000'). In FORTRAN, an integer value must be declared as INTEGER*4.

2. Scalar Real A SETL real is represented as a VAX longword; the value spans the full hardware range except for the special 'undefined real' value used by SETL for 'untyped real' (which has hex value '00000001'). In FORTRAN, a real value must be represented as REAL.

3. Tuple of Integer or Real A tuple of integers or reals is represented as an array of longwords. The tuple must contain no undefined elements, or 'holes'; ie., no element can be the undefined value OM. When passing values from SETL to the interface, the type of the tuple is determined from the type of the first component, which must be an integer or real. The remaining elements of the tuple must all have the same type. When returning values from the interface to SETL, all components of the tuple are interpreted according to the type specified.

4. Scalar String A SETL string is represented to FORTRAN as a character string.

7.2 SETL Interface Procedures

SETL procedure PUTF passes SETL values to a communication area. CALLF calls a FORTRAN procedure. GETF makes available results computed by the interface procedure.

7.2.1 PUTF(int,exp...exp)

The first integer argument of this new primitive gives the starting index in the communication tuple. The remaining arguments are then used to assign new values to the components of the communication tuple, starting at the specified index. The value of the first argument must be no greater than the length of the communication tuple, except that it can have value one greater than the length of the communication tuple (which is initially null), in which case new values are added to the communication tuple. The expression arguments of PUTF must all be defined (not OM).

Result values computed by FORTRAN procedures must have same type and structure as corresponding entry in communication tuple. In particular, output variables must have their type and structure indicated before CALLF is invoked. For example, if a procedure is to return an array of two integers, then the communication entry which it will use for the data which it returns must be initialized to the tuple [0,0].

7.2.2 CALLF(int,int,int)

The first argument of CALLF identifies the interface procedure be called. The second argument gives the starting index in the communication area at which the arguments of the FORTRAN procedure to be invoked have been placed, and the third argument gives the number of arguments to be passed. The components selected in the communication tuple are used to construct a standard FORTRAN argument list and the selected interface procedure is then called.

7.2.3 GETF(int,lhs...lhs)

The first argument to GETF identifies the starting index in the communication tuple from which values supplied by a FORTRAN procedure are to be read. Successive values are then retrieved from the communication tuple and assigned to the remaining arguments of GETF, in the order specified.

7.3 Using The Interface

To use the interface, you must supply a FORTRAN main program and auxiliary procedures which you wish to make available. Assume, to be specific, that all FORTRAN text is in file F.FOR, the SETL program in file F.STL. Compile your FORTRAN code to obtain F.OBJ, and your SETL program to get F.COD. Then link to obtain F.EXE as follows:

```
        $ LINK F+'STLLIB
```

Once this step has built your SETL extension, the resulting EXE file can be executed directly. You can also substitute your modified SETL for the standard SETL interpreter by executing the following assignment:

```
        $ SETL_LIB :== $DISK_SPEC:F
```

where DISK_SPEC must specify device and directory containing the file. (The full specification MUST be supplied due to the VMS conventions for defining 'foreign' commands.)

7.4 Sharing Of Data, Implementation Notes

The SETL system uses its own internal representation of values to maintain type information required for dynamic typing and to support dynamic storage management. The interface has been designed to pass values without needless copying; when necessary, copies of values are made in a work area maintained by the system. In general, copies in the work area are needed for all data types except strings and tuples of 'untyped' integers of reals. Values returned from the interface are always copied back into SETL dynamic storage.

NOTE

> Addresses of values passed from PUTF to the interface should not be saved by any FORTRAN procedure for use in a subsequent call to an interface procedure, as a SETL garbage collection may occur, thus voiding any saved addresses, which will then most likely refer to other values.

Aside from the cost of creating 'yet another SETL variant', the major cost of using this feature results from the need to include a full copy of the SETL run-time system in the executable program; this copy in turn includes a copy of the LITTLE system and system procedures themselves. These costs are avoided by default using the VAX/VMS system, which supports 'shared libraries'.

7.5 Sample Program

This section contains an example of the interface. Three features are provided by the SETL extension defined in our example:

1. String to integer conversion. This feature converts a string to an integer, and is useful since SETL VAL is not implemented.

2. Real to string formatting. SETL provides no formatting features for printing real values. This extension provides access to FORTRAN F-format formatting features.

3. Minimum element of tuple. This part of our example shows how to access and return tuple values.

A SETL program DEMO1.STL demonstrating this is as follows:

```
PROGRAM demo1;
$ This variant supports the following PUTF calls:
$ PUTF(1,STRING)
$   converts the second argument to integer
$ PUTF(2,REAL,INTEGER,INTEGER);
$  converts second argument to string using FORTRAN F format.
$  The third argument is the field length. The fourth argument
$  is the number of digits desired after the decimal point.
$  point.
$ PUTF(3,TUP)
$  searches the first argument, which must be integer tuple,
$  returns tuple whose first component is minimum value, and
$  whose second component is the index of the first element
$  in the input tuple having the minimum value.
$  If the input is the null tuple, the output is the tuple
[0,0].
LOOP DO
  PRINT('enter putf function number (1:3)');
  READ(n);
  IF EOF THEN QUIT; END IF;
  IF NOT IS_INTEGER n OR n<1 OR n>3 THEN CONTINUE; END IF;
  CASE n OF
    (1): PRINT('enter string');
         READ(s);
         PUTF(1,s,0); CALLF(1,1,2); GETF(2,r);
    (2): PRINT('enter real, field length, fract');
         READ(rv,fld,fract);
         PUTF(1,rv,fld,fract,' '*40); CALLF(2,1,4); getf(4,r);
    (3): PRINT('enter tuple');
         READ(tup);
         PUTF(1,tup,#tup,[0,0]); CALLF(3,1,3); GETF(3,r);
  END CASE;
  PRINT('result type, value: ', type r, r);
END LOOP;
END PROGRAM;
```

The FORTRAN program DEMO1.FOR defining the extension is as follows:

```
        external pimki,pifrs,pimin
        integer eara(3)
        eara(1) = %loc(pimki)
        eara(2) = %loc(pifrs)
        eara(3) = %loc(pimin)
        call STLINT(eara, 3)
        end
        subroutine pimki(str,iv)
!       convert input string to integer
        character*(*) str
        integer*4 iv
        call for$cnv_in_i(str, iv)
        return
        end
        subroutine pifrs(rv,length,fract,str)
!       convert real value to string
!       first arg is real, second arg is field length,
!       third arg is number of places after decimal point
        double precision dv
        real rv
        character*(*) str
        integer iv,fract,length
        dv = rv
        call for$cnv_out_f(dv, str,%val(fract),)
        return
        end
        subroutine pimin(itup,n,minr)
!       find minimum of integer tuple. Return tuple with first
!       component the minimum value, second component the least
!       index of input tuple with that value
        integer*4 minr(2)
        if (n.eq.0) then ! if null tuple
                minr(1) = 0
                minr(2) = 0
        else
        call pimin1(itup,n,minr)
        endif
        return
        end
        subroutine pimin1(ia,na,minr)
        integer*4 na,ia(na),minr(2)
        minr(1) = ia(1)
        minr(2) = 1
        do 1 i = 1,na
                if (ia(i).lt.minr(1)) then
                        minr(1) = ia(i)
                        minr(2) = i
                endif
 1      continue
        return
        end
```

The standard library procedures used in the preceding code are described in the VAX VMS Common Run-time Procedures Manual.

The demonstration program is built by the following command sequence:

```
        $ FORTRAN DEMO1  ! compile FORTRAN text
        $ SETL DEMO1/NORUN ! compile SETL text
        $ LINK DEMO1+'STLLIB
```

and is run by executing

```
        $ RUN DEMO1
```

In response to the prompt PARAMETERS produced by this last command, the user should type

```
        Q2=DEMO1.COD
```

Sample inputs are then solicited to verify and demonstrate the correct operation of the extension.

The program can also be used as the standard SETL library phase by executing the command

```
        $ setl_lib :== $file_spec:demo1
```

and then executing it using

```
        $ setlx demo1
```

8.0 Reserved Words

The following words have a predefined meaning within a SETL program, and should only be used for their defined purpose.

abs	false	match	real
acos	fix	max	remote
all	float	min	repr
and	floor	mmap	return
any	for	mod	rewind
arb	forall	mode	rmatch
asin	from	module	rnotany
assert	fromb	nargs	rpad
atan	frome	newat	rspan
atan2	general	not	rw
atom	get	notany	set
back	getb	notexists	setem
base	getem	notin	sign
boolean	getf	notrace	sin
break	getipp	npow	smap
callf	getk	odd	span
calls	getspp	of	sparse
case	goto	ok	sqrt
ceil	host	om	st
char	if	op	statements
close	impl	open	step
const	imports	operator	stop
continue	in	or	str
cos	incs	packed	string
date	init	pass	subset
debug	integer	plex	succeed
directory	is_atom	pow	tan
div	is_boolean	print	tanh
do	is_integer	printa	term
doing	is_map	proc	then
domain	is_real	procedure	time
drop	is_set	prog	title
eject	is_string	program	trace
elmt	is_tuple	put	true
else	len	putb	tuple
elseif	less	putf	type
end	lessf	putk	until
endm	lev	quit	untyped
eof	lib	random	val
error	libraries	range	var
even	library	rany	where
exists	local	rbreak	while
exit	loop	rd	with
exports	lpad	read	wr
expr	macro	reada	writes
fail	map	reads	yield

9.0 Implementation Dependent Information

Substantial work has been done to make the NYU LITTLE implementation portable so that implementations for different machines will be compatible. Some features, such as file names, command line format, and so forth, are necessarily machine dependent, and are described in this section.

9.1 DEC VAX-11 VMS Implementation

9.1.1 Configuration Requirements

This implementation runs on the Digital Equipment Corporation VAX-11 using the VMS V2 operating system. The operating system must have been configured with a value for VIRTUALPAGECNT not less than 8192, which provides for a per-process virtual address space of at least 4 megabytes.

9.1.2 Operating Instructions

Symbol definitions and command files for using SETL are available in file NYU$SETL:SETLDEF.COM. The easiest way to access them is to add

```
        $ @NYU$SETL:SETLDEF
```

to your LOGIN.COM file.

Individual phases may be run by using the symbolic names SETL_PRS, SETL_SEM, SETL_COD and SETL_LIB. However, for most applications the SETL and SETLX commands are more convenient.

Command SETL compiles and executes a SETL program. The form of the command line is:

```
        $ setl [sourcefile][inputfile][datafile][option...]
```

Sourcefile is the SETL source file. The sourcefile by default has extension "STL", so specification of this extension is unnecessary. Normally, the source file is compiled and executed. A code file is generated that has the name of the input file and the extension "COD".

The SETL command permits specification of the program parameters in standard VMS format.

/HARD
    Specifies that the code file is to be assembled into VAX machine code. The resulting executable file has extension "EXE".

/CODE[=file] (D)
/NOCODE

The code option indicates whether a code file is to be generated by the code generation phase (COD). The default is to generate such a file using the name of the input file and the default extension "COD". With the code option an explicit file may be specified; the default extension is "COD".

/DATA=file

Specifies the data file to be used in the interpreter phase (LIB). If not specified the data input file is assumed to be SYS$INPUT. Specification using this option overrides that specified by providing the datafile name to the SETL or SETLX commands. No default extension is provided. Any required extension must be given.

/LIST[=file]
/NOLIST (D)

The option /LIST is used to obtain a listing of the source file. If no file is specified, then the input file name together with the default extension "LIS" will be used. If an explicit listing file is given, the extension may be omitted, in which case the default, "LIS", will be used. The option /NOLIST signifies that no listing file is to be generated. If the program is executed then the listing will be written to the file specified, unless it is desired to have the execution output appear on a different file, in which case the XLIST option can be used to specify the file to receive the execution listing.

/PARM=

Specifies string to be included in parameter list passed to all compiler phases. If the string begins "NO", then these characters are removed, and the characters "=0" added at the end before passing along the argument. For example, /PARM=NOUV is translated to UV=0 which disables check for undeclared variables.

/RUN (D)
/NORUN

Specifies that the program is to be executed (interpreted).

/XLIST[=file]

Specifies the execution listing file, if it is desired that the execution listing be on a file different from that specified by the LIST option. If this option is not given, the default listing file is SYS$OUTPUT. When given, the default extension is "LIS". If /XLIST is specified without a file, then the source file name (SETL) or the input file name (SETLX) is used for the list file name, with extension "LIS".

/xxxFILE[=file]

This is a file specification parameter for SETL LIB execution. The 'xxx' denotes a one to three character extension, for example, 'DAT', the the file is assumed to have by default. The file specification may be absent, in which the source file name (SETL) or the input file name (SETLX) is used. The procedure call GETSPP('xxxFILE=/') may be used to interrogate this option within the program.

Command SETLX executes a previously compiled SETL program.

The form of the command line is:

```
       SETLX [codefile][inputfile][datafile][option...]
```

The code file is either a SETL 'code' file or an executable file with extension "EXE". If no explicit extension is given, then an executable file (with extension "EXE") is used if it exists; otherwise a code file with extension "COD" is assumed.

The input file is the standard SETL input file. If no file is specified, then SYS$INPUT is assumed. The data file is an alternate input file, identified through the SETL call GETSPP('DATA=/') in the program. If not specified, then SYS$INPUT is assumed.

In addition to any user supplied options, the following options described under the SETL command are relevant to SETLX:

```
    /ASSERT     /GTRACE     /H=n /LIST     /REL=n
    /SNAP       /STRACE     /TITLE /xxxFILE
```

9.1.3 Specifying Parameters

Program parameters for the SETL and SETLX command are specified in standard VMS fashion. The maximum length of the parameter list is 300 characters; the maximum length of a single parameter is 63 characters. When running the individual phases separately, the parameter list may be entered on the command line which invokes the program; if not entered, the program will prompt for parameters.
For example,

```
        $ SETL_PRS I=T.STL
```

9.1.4 Character Set

Full ASCII character set with upper and lower case letters.

9.1.5 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text. Instances of horizontal tabs and form feeds in the source are processed in the same way as blanks. The "such that" character ST can be represented using either vertical bar or the exclamation mark.

9.1.6 Input/Output

All input/output features are implemented. Text lines cannot exceed 132 characters.

On text output, trailing blanks and tabs are removed, except for files created using PUT.

The implementation has default PFPL=0/0 so that print file limits are not enforced by default.

9.1.7 Default File Names

Default file names are as follows:

I	SYS$INPUT/
ILIB	SYSLIB/
L	SYS$OUTPUT/
TERM	SYS$ERROR/
POL	POL.TMP/
XPOL	XPOL.TMP/
Q1	Q1.TMP/
Q2	Q2.TMP/
SQ1	0/SQ1.TMP
SSM	0/SSM.TMP

9.1.8 HOST Extensions

This implementation includes some special features provided by the HOST function.

NOTE

HOST features are not necessarily provided in other implementations of SETL. Also, these features may change or disappear as new releases of SETL appear.

The features are grouped into related functions, called "packages", and the first argument of HOST is an integer used to select the package. The packages are as follows:

9.1.8.1 Device-dependent Terminal Input/Output

HOST(1, 1)

returns a string of length one containing the next character typed at the terminal

HOST(1,2,str...str)

writes strings to terminal

No extra characters are added or removed, so that user must issue carriage returns, line feeds, as needed. All characters can be read except control-s, control-q, control-y, control-c, and control-o, which are intercepted and processed by the terminal driver in the usual way. Note that this is "raw" i/o and problems may arise if other means are used to communicate with the terminal. For example, using these procedures and PRINT procedure will probably cause problems.

Useful initialization statements that may be needed:

```
 ccbel := char 7; $ bell
 ccbs  := char 8; $ backspace
 ccht  := char 9; $ horizontal tab
 cclf  := char 10; $ line feed
 cccr  := char 13; $ carriage return
 ccesc := char 27; $ escape
```

For example, to send line with normal carriage control to terminal, use

```
  HOST(1,2,line,cccr,cclf);
```

9.1.8.2 Substring Search

HOST(2, str, str)

Searches the third argument for an instance of the second argument. Yields the position of the first (leftmost) occurrence if the search succeeds, or yields zero if no occurrence found. Yields zero if the second or third argument is the null string.

9.1.8.3 Case Conversion

HOST(3, int, str)

Converts case of third argument according to value of second argument. If second argument is zero, the result is the third argument converted to lower case; otherwise, the result is the third argument converted to upper case.

9.1.8.4 Execute Commands

HOST(4, str, str ... str)

The argument strings are executed in a subprocess using the system service program LIB$SPAWN.

For example,

```
 HOST(4,'setl x'); $ compiles program
 HOST(4,'dir/out=d.out');  $ gets directory
```

N.B. each 'str' is spawned as a separate subprocess.

9.1.8.5 Device-independent Terminal Input/Output

These procedures permit device-independent communication with the terminal. The screen is addressed by giving a line number and a column number. The top line has line number one; the first character in a line has column number one.

These procedures use the VMS Run-time Library procedures in the "Terminal Independent Screen Procedures" package described in Section 3.2 of the Run-time Library Manual. The terminal types supported are the standard DEC types VT52 and VT100, as well as any "foreign" terminals supported by your site (see your system manager for a list of such types; they are types known to VMS SET TERMINAL/FT command.) These functions can be used with non-terminal devices, in which case cursor control features are ignored.

HOST(5, 0, string)

Displays prompt given by third argument, then reads line from the terminal (terminated with RETURN), and yields the string read in.

HOST(5, 1, string)

Displays the third argument.

HOST(5, 2)

Erases the screen to end of page.

HOST(5, 3)

Erases the screen to end of line.

HOST(5, 4, integer, integer)

Positions the cursor at line given by third argument, column given by fourth argument.

HOST(5, 5)

Does "reverse index". The cursor is moved up one line, unless it is in the top line, in which case all lines are moved down one line, the top line is replaced with a blank line and the the data that was on the bottom is lost.

9.1.9 'Mapped Heap Files'

The compiler for VAX/VMS produces a representation of the program in a form known as 'Q2' which is interpreted. Interpreted execution begins by reading in miscellaneous variables and the initial contents of the SETL heap, which also includes the Q2 code. These values are read in using binary input. For large programs, especially when run interactively, there is a delay since the entire intermediate text must be read in before interpreted execution can begin.

VMS permits the user to define disk files containing the initial data for a program in a form which can be 'mapped' into the program virtual address space; indeed, this is the mechanism used for the executable files produced by the loader. This note outlines a simple scheme for using the mapping primitive to effect more efficient initialization of the execution phase of SETL programs.

The basic idea is to split the standard Q2 file into two files: the 'Q2H' file contains the heap data in a form which permits direct mapping to virtual memory using the VMS system service 'crmpsc'. The 'Q2E' file consists of the other data in the Q2 file.

To initialize using a mapped heap file, the Q2E file is used instead of the standard Q2 file and the Q2H file is specified.

9.1.9.1 Relevant Program Parameters

The following new program parameters are recognized by SETL LIB

Q2INIT=0/1	initialization type
Q2E=Q2E/	Q2E file name
Q2H=Q2H/	Q2H file name
HFTRACE=0/1	nonzero to trace hf procedures

9.1.9.2 Usage

The Q2INIT option determines how heap initialized:

Q2INIT=0
: Read in heap data from Q2 file (default)

Q2INIT=1
: Map heap data from file specified by Q2H option; read other data from Q2 file (actually Q2E file produced in case 2).

Q2INIT=2
: Read in Q2 file, create mapped heap file and write modified Q2 file to files specified by Q2H and Q2E options, respectively. Execution terminates after these files created.

Example:

Given program T.STL, compile to get T.COD. Then to get Q2H and Q2E files, do

```
  $ SETLX T/Q2INIT=2/Q2E=T.Q2E/Q2H=T.Q2H
```

To execute using Q2H and Q2E files:

```
  $ SETLX T.Q2E/Q2INIT=1/Q2H=T.Q2H/Q2E=T.Q2E
```

9.1.10 SETL Hard Code System

The hard code system makes possible more efficient execution of SETL programs by producing a machine language file which can be directly executed. This alternate means of effecting execution emphasizes execution speed at the expense of space; the hard code version should run faster, though more space will usually be needed.

To use the hard code system, a program must be compiled using the HARD qualifier. This will produce an executable file (with type .EXE) as output and also a Q2 file. The executable file is complete and may be used in place of the SETL interpreter. The format of the Q2 files produced by standard compilation and compilation with hard code is incompatible; a Q2 file produced by hard code compilation cannot be used with the interpreter, and a Q2 file produced by standard compilation cannot be used with a hard code .EXE file. For VAX/VMS, the system is used as follows. First, compile using the HARD qualifier:

```
        $ setl X/hard/norun
```

This will produce X.EXE and X.COD. The SETLX command knows about .EXE files and will use such a file if it exists.

The option SIF may be specified to retain the intermediate files produced by the translation. The program parameters related to the hard code system are as follows:

HQ2F=0/1
: Controls whether the intermediate listing file, of type .HCL, contains the static frequency of the Q2 opcodes in the Q2 (.COD) file.

ASMTR=0/1
: Controls whether the ASM phase of the hard code translation generates debugging output.

HXSTMT=0/1
: Controls whether the code generated for Q2 STMT opcodes is entered and executed. The default is not to enter the code. Specify HXSTMT=1 to execute the full statement code; this specification must be made if you want to use features such as STRACE or if you want the statement number correctly reported in the event of an error.

The parameters of most interest to the ordinary user are STMT and HXSTMT. The Q2 form of the program includes a STMT opcode to record the position within a program; this position is used to report the point at which an error occurred, and also by such features as the STRACE option which traces statements as they are executed. However, the overhead of keeping track of the position can be large (it has been observed to consume about 20 percent of the execution time of some programs); and the hard code system permits a more efficient processing of the STMT opcodes. The setting of the STMT parameter determines if any code to support STMT opcodes is generated. You should use STMT=0, causing no code to be generated, only if you are interested in maximum performance and will never want to use any of the features related to the STMT opcodes.

The HXSTMT parameter determines the extent to which STMT opcodes are recognized at runtime. Of course, if STMT=0 was specified when the file was translated by the hard code system, then the setting of the HXSTMT parameter is unimportant, as the STMT opcodes have already been eliminated. Given that STMT opcodes have been translated, then the value of the HXSTMT parameter is kept in a register and each STMT opcode is translated into a test of the HXSTMT value (which is kept in a register) so that if HXSTMT=0 is specified the the full code to process the STMT opcode is branched around. To use features dependent on the STMT opcode, HXSTMT=1 must be specified.

The default is HXSTMT=0, so that STMT opcodes are branched around. If your program terminates abnormally and you want to find where the error occurred, repeat execution with the specification HXSTMT=1.

9.1.11 Restrictions

1. Real arithmetic restricted to single (long word) precision.

2. At most 65535 elements in set, tuple or character string.

9.2 DEC VAX-11 UNIX Implementation

9.2.1 Configuration Requirements

This implementation runs on the Digital Equipment Corporation VAX-11 using the Berkeley 4.2 BSD UNIX operating system.

```
NAME
        stl - Setl compiler and interpreter
SYNOPSIS
        stl -{cox} [-l] [-O] [-v] file [programparameters]
DESCRIPTION
Stl invokes the NYU Setl compiler and interpreter.  Stl -c takes a  Setl
source file, suffixed `.stl', and produces a listing file `.lis', and an
initial run-time environment file, suffixed `.cod', which is interpreted
by  stl  -x.   Setl-specific compile and run-time program parameters are
supplied after the file name.  A list  of  the  program  paramterers  is
given in the Setl user manual.
FILES
        /usr/local/stl* (see definition of $SETL there)
        $SETL/user.doc, user manual
SEE ALSO
        ``The SETL Programming Language'', R. B. K. Dewar
        ``Higher Level Programming'', R. B. K. Dewar, E. Schonberg,
        J. T. Schwartz
BUGS
        Only the first 240 characters of the command line are
        examined by the SETL system.
```

9.2.2 Specifying Parameters

Program parameters are specified in standard fashion. The maximum length of the parameter list is 240 characters; the maximum length of a single parameter is 63 characters.
For example,

```
        stl -c t.stl list reprs=1
        stl -x t.cod
```

9.2.3 Character Set

Full ASCII character set with upper and lower case letters.

9.2.4 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text. Instances of horizontal tabs and form feeds in the source are processed in the same way as blanks. The "such that" character ST can be represented using either vertical bar or the exclamation mark.

9.2.5 Input/Output

All input/output features are implemented.

On text output, trailing blanks and tabs are removed, except for files created using PUT.

The implementation has default PFPL=0/0 so that print file limits are not enforced by default.

9.2.6 Default File Names

UNIX has no notion of file name in the SETL sense, so the implementation uses the following 'names' for the standard unix files: stdin, stdout and stderr. Default file names are as follows:

i	stdin/
ilib	syslib/
l	stdout/
term	stderr/
pol	pol.tmp/
xpol	xpol.tmp/
q1	q1.tmp/
q2	q2.tmp/

```
 sq1    0/sq1.tmp
```

9.2.7 Restrictions

1. Real arithmetic restricted to single (long word) precision.
2. At most 65535 elements in set, tuple or character string.

9.3 Amdahl UTS Implementation

9.3.1 Configuration Requirements

This implementation runs on the 370 architecture using the Amdahl Corporation UTS operating system.

```
NAME
        stlc, stl - setl compiler and interpreter
SYNOPSIS
        stlc file.stl [options]
        stl file.cod [options]
DESCRIPTION
Stlc and stl form the SETL system.  Stlc  takes  a  SETL  source  file,
suffixed  `.stl'  and produces a listing file `.lis', and an object file
`.cod', which is given  to  stl.   Compile  and  run-time  switches  are
supplied  after  the  file name.  A list of the switches is given in the
SETL user manual.
FILES
        /usr/local/stl* (see definition of $SETL there)
        $SETL/user.doc, user manual
SEE ALSO
        ``The SETL Programming Language'', R. B. K. Dewar
BUGS
        Only the first 240 characters of the command line are
        examined by the SETL system.
```

9.3.2 Specifying Parameters

Program parameters are specified in standard fashion. The maximum length of the parameter list is 240 characters; the maximum length of a single parameter is 63 characters.
For example,

```
        stlc t.stl list reprs=1
        stl  t.cod
```

9.3.3 Character Set

Full ASCII character set with upper and lower case letters.

9.3.4 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text. Instances of horizontal tabs and form feeds in the source are processed in the same way as blanks. The "such that" character ST can be represented using either vertical bar or the exclamation mark.

9.3.5 Input/Output

All input/output features are implemented.

On text output, trailing blanks and tabs are removed, except for files created using PUT.

The implementation has default PFPL=0/0 so that print file limits are not enforced by default.

9.3.6 Default File Names

UTS has no notion of file name in the SETL sense, so the implementation uses the following 'names' for the standard unix files: stdin, stdout and stderr. Default file names are as follows:

i	stdin/
ilib	syslib/
l	stdout/
term	stderr/
pol	pol.tmp/
xpol	xpol.tmp/
q1	q1.tmp/
q2	q2.tmp/
sq1	0/sq1.tmp

9.3.7 Restrictions

1. Integer arithmetic restricted to single long word operands. Integer arithmetic is correct in the range -2**31+1 to 2**31-1. The hardware value -2**31 is reserved for the "undefined" integer, for example, (1 div 0) yields this value.

2. Real arithmetic restricted to single precision.

3. At most 65535 elements in set, tuple or character string.

9.4 CDC 6000 Implementation

9.4.1 Configuration Requirements

This implementation runs on the Control Data Corporation 6000 Series hardware. It can be configured for NOS or NOS/BE operating systems, 63 or 64 character set.

The PRS, SEM and COD phases are combined into a single program SETL which requires about 170000B to run. The LIB phase requires 170000B words plus the SETL heap.

9.4.2 Operating Instructions

Needed files are kept in directory SETL. The control statements to compile and execute program on file SETLI, with listing, are as follows:

```
        ATTACH,SETL,STLLIB/UN=SETL.
        SETL. (I=SETLI,LIST)
        STLLIB.
```

9.4.3 Specifying Parameters

Program parameters are NOT specified in the usual CDC fashion, but are given in a separate list which follows program name. Parameters are enclosed within parentheses and separated by commas. Note that

```
        SETL(I=SETLIN)
```

is WRONG. The correct specification is:

```
        SETL. (I=SETLIN)
```

9.4.4 Character Set

DISPLAY code. For 64 set sites, the per-cent character can be used where colon required. The following graphics (selected by the program parameter CSET=EXT, which is the default) are used:

SETL	DISPLAY code
{	74 octal (at sign)
}	75 octal (reverse slant)
ST	67 octal (ampersand)

9.4.5 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text, and lists 90 columns to permit use with UPDATE.

9.4.6 Default File Names

Default file names are as follows:

I	INPUT/COMPILE
ILIB	INCLIB/
L	OUTPUT/LIST
TERM	/TERM
POL	POL/POL
XPOL	XPOL/XPOL
Q1	Q1/Q1
Q2	Q2/Q2
SQ1	0/SQ1

9.4.7 Stand-alone Parse

Due to large size of the SETL system, your site may have installed a variant of SETL PRS which does only syntax analysis, and has smaller internal tables, but which can be run using about 100000B words. If available, this is used as follows:

```
        ATTACH,SETLP/UN=SETL.
        SETLP.
```

9.4.8 Storage Allocation

Due to large size of SETL system, users of the 6000 version may find it necessary to force the system to use a small initial dynamic memory area (heap). The heap is divided into three regions: a "constants" area which contains constants and the code, the run-time stack and the remainder which is used for values built during execution.

The program parameters H, ST and CA can be used to determine initial structure of the heap. The defaults are:

H=8000/8000	total length
ST=0/0	symbol table length
CA=0/0	constants area length

Specifying a small integer value less than 1024 causes multiplication by 1024; for example H=4 taken as H=4096. Specifying 0 directs the system to use reasonable guesses. At present the guesses are to allocate H/2 words for the constant area, and H/8 words for the symbol table. These guesses can be avoided by providing nonzero values for ST and CA parameters. For small programs, H=4 is suggested.

The listing file produced by the COD phase indicates the actual lengths of the constant area and symbol table, and should be consulted for hints on picking parameter values.

9.4.9 Restrictions

1. Integer arithmetic restricted to single word operands.
   Integer arithmetic is correct in the range -2**48-1 to 2**48-1.

2. Real arithmetic restricted to single precision.

3. At most 32767 elements in set, tuple or character string (if enough memory available!).

9.5 DEC DECsystem-10 Implementation

9.5.1 Configuration Requirements

This implementation runs on the Digital Equipment Corporation DECsystem-10 hardware using the TOPS-20 operating system. The implementation should also be usable on TOPS-10 and TENEX, although this has not been verified.

9.5.2 Operating Instructions

At Rutgers, using TOPS-20, SETL is currently available on s:<setl.final>. The phases of the compiler should be run in turn. For example, to compile and execute X.STL, proceed as follows:

```
        def sys: s:<setl.final>,sys:
        stlprs(i=x.stl)
        stlsem(i=x.stl)
        stlcod(i=x.stl)
        stllib(i=x.stl)
```

9.5.3 Specifying Parameters

Program parameters are specified in the usual LITTLE fashion, i.e., as list enclosed in parentheses following program name. The I= parameter should always be specified, even if a dummy file must be created; for example,

```
        stlsem(i=foo.stl)
```

The maximum length of the parameter list is 120 characters; the maximum length of a single parameter is 30 characters. When running the individual phases separately, the parameter list may be entered on the command line which invokes the program; if not entered, the program will prompt for parameters. For example,

```
        $ run stlprs
```

Note that the parameter line is converted to upper case. This is generally not significant. However, arguments to the procedures GETIPP and GETSPP should thus be specified in upper case. For example,

```
        TRVAL := GETIPP('TRACE=0/1');
```

9.5.4 Character Set

Full ASCII character set with upper and lower case letters.

9.5.5 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text. Instances of horizontal tabs and form feeds in the source are processed in the same way as blanks. The "such that" character ST can be represented using either vertical bar or the exclamation mark.

9.5.6 Input/Output

The input/output procedures GET and PUT are not implemented Text lines cannot exceed 132 characters. On text output, trailing blanks and tabs are removed.

9.5.7 Default File Names

Default file names are as follows:

I	*.LTL/*.LTL (however, see section on program parameters below)
ILIB	SYSLIB/SYSLIB
L	*.LST/*.LST
TERM	TTY:/
POL	POL/POL
XPOL	XPOL/XPOL
Q1	Q1/Q1
Q2	Q2/Q2
SQ1	0/SQ1

Note that * indicates that name given by I= parameter is used to derive filename and extent is then chosen based on at most first three characters of parameter values as shown above.

9.5.8 Restrictions

1. Integer arithmetic restricted to single word operands.
   Integer arithmetic is correct in the range -2**35+1 to 2**35-1.
   The hardware value -2**35 is reserved for the "undefined" integer, for example, 1/0 yields this value.

2. Real arithmetic restricted to single (long word) precision.

3. At most 65535 elements in set, tuple or character string.

The following features are not implemented:

1. exponentiation

2. Mathematical functions SQRT, ATAN, ATAN2, COS, SIN, EXP, LOG, TAN and TANH.

3. DATE is implemented but the day of week is always Wednesday.

9.6 IBM System/370 CMS Implementation

9.6.1 Configuration Requirements

This implementation runs on the International Business Machines Corporation System/370 hardware. It is configured for the CMS operating system; it should be usable using OS and its extensions (MVS, etc.), though usage for these systems has not been tested.

The PRS, SEM and COD phases are combined into a single program SETL. The LIB phase is available as STLLIB.

9.6.2 Operating Instructions

Needed files are kept on a minidisk of user SETL. See the system manager for information about accessing this disk.

The control statements to compile and execute program on file SETLI SETL A1, with listing on file SETLI LISTING A1, are as follows:

```
        SETL SETLI (LIST RUN
```

The single (required) operand of the SETL command is a file identifier (SETLI in the previous example) of the form:

```
        fn ft fm
```

The default ft is the name of the program (SETL in this example). The default fm is A1.

9.6.3 Specifying Parameters

Program parameters are entered as CMS options. However, to overcome the CMS limitation of eight characters per argument, the parameter scanner also does the following:

1. Blanks not following an equal sign are taken as commas.

2. Blanks just after an equal sign are ignored.

As a result, the following are equivalent:

```
        SETL SETLI (LIST, H=4
        SETL SETLI (LIST H=4
        SETL SETLI (LIST H= 4
```

Note that the (added) parameter RUN causes SETL LIB to execute the program once it has been compiled. To execute an already compiled, program, use the command:

```
        STLLIB PROG (I=0
```

9.6.4 Character Set

EBCDIC with upper and lower case letters. The following graphics (selected by the program parameter CSET=EXT, which is the default) are used:

SETL	EBCDIC code
{	8B hex
}	9B hex
ST	4F hex (vertical bar)
ST	5A hex (exclamation mark)
[	AD hex
]	BD hex

Lower case letters and non-standard graphics are recognized, but are not generated in normal operation; they are generated only at the explicit request of the user, or as a result of copying lower case characters in source and data files.

9.6.5 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text, and lists 80 columns to display any sequence information in positions 73..80.

9.6.6 File Names

The file names used by SETL (and specified as parameters) are DDNAMEs. If an explicit FILEDEF has been given for the DDNAME, it is used. Consistent with the normal conventions for OS compilers running under CMS, the following DDNAMEs are translated in the absence of a FILEDEF for them:

SYSPRINT	to LISTING
SYSPUNCH	to PUNCH
SYSTERM	to TERM
SYSUTn	to CMSUTn

If an explicit FILEDEF is specified, it will be used. Otherwise, an implicit FILEDEF will be executed. This implicit FILEDEF will be of the form:

```
        FILEDEF ddname DISK fn ddname A1
```

where fn is the filename of the operand of the command.

If no FILEDEF is specified for SYSIN, the following is done:

```
        FILEDEF SYSIN DISK fn ft fm
```

where fn ft fm are the components of the operand with the defaults supplied as described above.

There are exceptions to the implicit FILEDEF described above. These are ddnames of TERMx, PRINT and PUNCH. In these cases the device represented by the ddname specified will be used, i.e., the following FILEDEF will be executed:

```
        FILEDEF ddname ddname
```

Default file names are as follows:

I	SYSIN/SYSIN
ILIB	SYSLIB/SYSLIB
L	SYSPRINT/SYSOUT
TERM	SYSTERM/SYSTERM
POL	POL/POL
XPOL	XPOL/XPOL
Q1	Q1/Q1
Q2	Q2/Q2
SQ1	0/SQ1

9.6.7 Restrictions

1. Integer arithmetic restricted to single word operands.
   Integer arithmetic is correct in the range -2**31+1 to 2**31-1.

2. Real arithmetic restricted to single precision.

3. At most 32767 elements in set, tuple or character string.

9.7 IBM System/370 MTS Implementation

9.7.1 Configuration Requirements

This implementation runs on the International Business Machines Corporation System/370 hardware. It is configured for the MTS operating system, but since no work has been done after the initial bootstrap, not all looks the way one might like to see it.

9.7.2 Operating Instructions

Needed files are kept under user SETL. See the system manager for information about accessing these files.

The PRS, SEM and COD phases are executed as separate phases: to compile a SETL program in file inFDname into a file codFDname with a listing into file listFDname, do the following:

```
    $ set libsrch=stllib+ltllib
    $ run stlprs par=i=inFDname,l=listFDname,term=0,list,at
    $ run stlsem par=i=0,l=listFDname(*l+1),term=0
    $ run stlcod par=i=0,l=listFDname(*l+1),term=0,q2=codFDname
```

To execute this program, type:

```
    $ set libsrch=stllib+ltllib
    $ run stlint par=i=inputFDname,l=outputFDname,q2=codFDname
```

where inputFDname is the standard input file (e.g. *source*), and outputFDname is the standard output file (e.g. *sink*). At the moment, the default file names are the OS file names, i.e. SYSIN and SYSPRINT. Likewise, the default for the 'terminal' file is SYSTERM.

9.7.3 Specifying Parameters

Program parameters are entered at the end of the PAR-string.

9.7.4 Character Set

EBCDIC with upper and lower case letters. The following graphics (selected by the program parameter CSET=EXT, which is the default) are used:

SETL	EBCDIC code
{	8B hex
}	9B hex
ST	4F hex (vertical bar)
ST	5A hex (exclamation mark)

[	AD hex
]	BD hex

Lower case letters and non-standard graphics are recognized, but are not generated in normal operation; they are generated only at the explicit request of the user, or as a result of copying lower case characters in source and data files.

9.7.5 Source Program Format

The compiler examines only the first 72 columns of each line of SETL source text, and lists 80 columns to display any sequence information in positions 73..80.

9.7.6 Input/Output

All input/output features are implemented.

9.7.7 Default File Names

The file names used by SETL (and specified as parameters) are FDnames. Default file names are as follows:

I	SYSIN/
ILIB	SYSLIB/
L	SYSPRINT/SYSOUT
TERM	SYSTERM/
POL	-SETLPOL/
XPOL	-SETLXPOL/
SSM	-SETLSSM/
Q1	-SETLQ1/
Q2	Q2/
SQ1	0/-SETLSQ1

9.7.8 Restrictions

1. Real arithmetic restricted to single precision.

2. At most 65535 elements in set, tuple or character string.

APPENDIX A

SEMANTIC DEFINITIONS OF STRING PRIMITIVES

This section contains the semantic definitions of the string search primitives, in the form used in chapter 7 of the SETL reference manual.

```
PROC ANY(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a > 0 AND #b > 0 AND a(1) IN b THEN
      t := a(1);
      a := a(2..);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC ANY;

PROC BREAK(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF EXISTS i IN [1..#a] ST a(i) IN b THEN
      t := a(1..i-1);
      a := a(i..);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC BREAK;

PROC LEN(RW a,b);
  CASE OF
  (IS_STRING a AND IS_INTEGER b AND b>=0):
    IF b<#a THEN
      RETURN OM;
    ELSE
      t := a(1..b);
      a := a(b+1..);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC LEN;

PROC LPAD(a,b);
  CASE OF
  (IS_STRING a AND IS_INTEGER b):
    IF b < #a THEN
      RETURN a;
    ELSE
      RETURN ' ' * (b - #a) + a;
    END;
  ELSE /*error*/;
  END CASE;
END PROC LPAD;

PROC MATCH(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a >= #b AND a(1..#b) = b THEN
      a := a(#b+1..);
      RETURN b;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC MATCH;

PROC NOTANY(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a > 0 AND #b > 0 AND a(1) NOTIN b THEN
      t := a(1);
      a := a(2..);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC NOTANY;

PROC RANY(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a > 0 AND #b > 0 AND a(#a) IN b THEN
      t := a(#a);
      a := a(1..#a-1);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC RANY;

PROC RBREAK(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF EXISTS i IN [#a,#a-1..1] ST a(i) IN b THEN
      t := a(i+1..);
      a := a(1..i);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC RBREAK;

PROC REPLACE(a,b,c);  $ not implemented
  CASE OF
  (IS_STRING a AND IS_STRING b AND IS_STRING c AND #b = #c):
    t := '';
    (FOR d IN a)
      IF EXISTS i in [#b,#b-1..1] ST d = b(i) THEN
        t +:= c(i);
      ELSE
        t +:= d;
      END IF;
    END;
    RETURN t;
  ELSE /*error*/;
  END CASE;
END PROC REPLACE;

PROC REVERSE(a);        $ not implemented
  CASE OF
  (IS_STRING a):
    RETURN '' +/[a(#a-i+1): i IN [1..#a]];
  ELSE /*error*/;
  END CASE;
END PROC REVERSE;

PROC RLEN(RW a,b);
  CASE OF
  (IS_STRING a AND IS_INTEGER b AND b>=0):
    IF b<#a THEN
      RETURN OM;
    ELSE
      t := a(#a-b+1..);
      a := a(1..#a-b);
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC RLEN;

PROC RMATCH(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a >= #b AND a(#a-#b+1..) = b THEN
      a := a(1..#a-#b);
      RETURN b;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC RMATCH;

PROC RNOTANY(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a > 0 AND #b > 0 AND a(#a) NOTIN b THEN
      t := a(#a);
      a(#a) = OM;
      RETURN t;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC RNOTANY;

PROC RPAD(a,b);
  CASE OF
  (IS_STRING a AND IS_INTEGER b):
    IF b < #a THEN
      RETURN a;
    ELSE
      RETURN a + ' ' * (b - #a);
    END;
  ELSE /*error*/;
  END CASE;
END PROC RPAD;

PROC RSPAN(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a > 0 AND #b > 0 THEN
      IF {x: x IN b} INCS {x: x IN a} THEN  $ if span all
        t := a;
        a := '';
        RETURN t;
      ELSEIF EXISTS i IN [#a,#a-1..1] ST a(i) NOTIN b THEN
        t := a(i+1..);
        a := a(1..i);
        RETURN t;
      ELSE
        RETURN OM;
      END IF;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC RSPAN;

PROC SPAN(RW a,b);
  CASE OF
  (IS_STRING a AND IS_STRING b):
    IF #a > 0 AND #b > 0 THEN
      IF {x: x in b} INCS {x: x IN a} THEN $ if span all
        t := a;
        a := '';
        RETURN t;
      ELSEIF EXISTS i IN [1..#a] ST a(i) NOTIN b THEN
        t := a(1..i-1);
        a := a(i..);
        RETURN t;
      ELSE
        RETURN OM;
      END IF;
    ELSE
      RETURN OM;
    END;
  ELSE /*error*/;
  END CASE;
END PROC SPAN;
```

APPENDIX B

CHANGES EFFECTED IN RECENT VERSIONS

Changes from Version 29 to Version 30:

1. System Internal changes which should be transparent to the user.

2. UNIX Implementations suppress by default the compiler listing of program parameters and compilation statistics. They can be controlled by the LCP and LCS program parameters.

3. Fix several small bugs.

Changes from Version 28 to Version 29:

1. Fix several small bugs.

Changes from Version 27 to Version 28:

1. Support Motorola 68000 Microprocessor on SUN Workstation.

2. Fix several small bugs.

Changes from Version 26 to Version 27:

1. The semantics of some qualifiers has been changed:

```
/ASSERT=0: Expressions appearing in assert statements are
           not evaluated.
           (These expressions were evaluated before.)
/MAX_HEAP=n Allow heap expansion to 'n' SETL words
           (default: 512000).
/REPRS=2:  Equivalent to /REPRS[=1];  in addition, a
           message is printed for each conversion generated
           between non-primitive values.
           /REPRS=2 requires the /LIST qualifier to have
           any effect.
```

2. Integer arithmetic supports integers in the range

```
- ((32768 ** 65535) - 1) .. (32768 ** 65535) - 1.
```

on 32-bit machines. Integer denotations are restricted to the range -2147483647..2147483647. The result of constant expressions is restricted to the range of integer denotations, i.e. -2147483647..2147483647.

3. The MAP representation is fully supported.

4. The semantics of the assert statement 'assert <lhs> := <rhs>;' has been changed to mean 'test whether <lhs> = <rhs>; if they are unequal, print an error message and assign <rhs> to <lhs>.' (cf. SN210) (Previously, this would assign <rhs> to <lhs> and test the value of <lhs> for true/false.)

5.

The Q2 (code)- and binary file formats have been changed. All Q2- and binary files must be recreated under this version of SETL.

Changes from Version 25 to Version 26: The Q2 (COD) format has (once again) changed.

Changes from Version 24 to Version 25:

1. The implementation of the mode 'integer i..j' has been changed: if i > 0, the old implementation widened the range to 'integer 1..j', the new implementation does exact range analysis within a machine-dependent range, currently for 32-bit machines:

   0 <= i < 256, 0 <= j < 65535.

2. The mode 'map' and 'map (<mode>) <mode>' is recognized by the front-end once more, yet might produce incorrect run-time results. An appropriate warning is printed whenever this mode is used. THE MODE 'MAP' SHOULD NOT BE USED UNTIL FURTHER NOTICE.

3. Add the STMT program parameter.

4. Implement "string IN string" and "string NOTIN string" using an algorithm, due to Knuth, that is linear in the length of the two strings.

Changes from Version 23 to Version 24:

1. Eliminate null statement.

2. Allow representation declaration 'proc() mode' for procedures with no parameters.

3. MAP is now a reserved word.

4. Add program parameter UR.

5. Change default for program parameter REPRS.

6. Reserved words cannot be used as member names.

7. The interface statement 'READS ALL' is no longer the default; all variables and constants that are accessed must be declared.

Changes from Version 22 to Version 23:

1. Permit second argument of OPEN to be written in either case, so long as all characters in a particular use are in the same case.

2. Add SOCASE option.

3. Add TERMP and TERMH options.

4. Recognize NOTEXISTS.

5. Precedence of ? operator: the precedence of the query operator is 10. Therefore, if either operand is an expression, the usual precedences are applied. If in doubt, use parentheses.

6. The precedence of <exp> <*binop> '/' <exp> has been corrected, so that 2 * 1 +/ [2] evaluates to 4, and not to 6.

7. The parser now accepts '(' <exp> ')' <index*>, so that (a)(b) is a valid expression.

8. The arithmetic iterator has been changed to include a test for zero increment, where required. Thus the cardinality of [m, m .. n] is zero, and not an infinite loop.

Changes from Version 21 to Version 22 (not all of which are yet documented at proper place):

1. FORTRAN Interface. It is now possible to create user-defined extension permitting invocation of FORTRAN procedures during execution. This is currently only available for VAX/VMS.

2. Mapped Q2 file. The VAX/VMS version permits creation of special files which permit quicker program startup for large programs. This involves new parameters NOF, Q2INIT, Q2E, Q2H and HFTRACE.

3. %CMODE (VAX/VMS). At execution the program parameter %CMODE is set to indicate how a program is being run. After

```
        cmode := GETSPP('%CMODE=/');
```

then cmode will be either 'INTERACTIVE' or 'BATCH' with obvious meaning.

4. MACRO local symbols. There are restrictions on the use of local symbols in macros not yet documented. (This is a warning, not a change.)

5. HOST(1..) (VAX/VMS) This does not report end of input correctly.

6. SPEC and UNSPEC are no longer reserved words.

INDEX