The SETL2 Programming Language

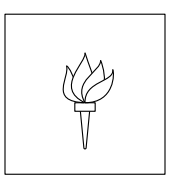
W. Kirk Snyder
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012
September 9, 1990

Abstract

The SETL2 programming language is a very high level language based on the theory and notation of finite sets. It is evolved from SETL, developed at New York University by J. T. Schwartz. SETL2 adds to SETL a syntax and name scoping closer to more recent imperative languages, full block structure, and procedures as first class objects.

This document is divided into three parts: First, we give an introduction to SETL2, highlighting its differences from SETL. A description of programming with sets along with a detailed description of SETL is given in [SDDS86], and we strongly encourage the reader to refer to that book if he has no experience with SETL. The second part of this document provides a detailed reference manual of the SETL2 language. The final section is a guide to the operation of the current implementations of SETL2.

This paper is Technical Report 490, Courant Institute of Mathematical Sciences, New York University.



New York University
Department of Computer Science
Courant Institute of Mathematical Sciences

Contents

1 An Introduction to SETL2 **1**

1.1 Overview . . . 1

1.2 Nested Procedures . . . 1

1.2.1 Name Scoping . . . 1

1.3 Procedures As First Class Objects . . . 2

1.3.1 Write Parameter Restriction . . . 3

1.3.2 Closures . . . 3

1.4 Anonymous Procedures – λ Expressions . . . 4

1.5 Separate Compilation – Packages . . . 6

1.6 Bound Variables Are Local To Iterators . . . 7

1.7 Ada-Like Syntax . . . 8

1.8 Summary . . . 8

2 SETL2 Reference Manual **9**

2.1 Introduction . . . 9

2.2 Notation and Terminology . . . 9

2.3 Lexical Conventions . . . 9

2.3.1 Comments . . . 9

2.3.2 Identifiers . . . 9

2.3.3 Numeric Literals . . . 10

2.3.4 String Literals . . . 10

2.3.5 Operators . . . 11

2.3.6 Separators . . . 11

2.4 Overall Program Structure . . . 11

2.5 Declarations and Scope of Names . . . 12

2.5.1 Selectors . . . 13

2.6 Statement Lists . . . 13

2.7 Procedures . . . 13

2.7.1 Return Statements . . . 14

2.8 Basic Data Types . . . 15

2.8.1 Integers . . . 15

2.8.2 Floating Point Numbers . . . 15
2.8.3 Strings . . . 16
2.8.4 Atoms . . . 16
2.8.5 Procedures . . . 16
2.8.6 Ω - Undefined Value . . . 16
2.9 Compound Data Types . . . 17
2.9.1 Sets . . . 17
2.9.2 Tuples . . . 17
2.9.3 Maps . . . 18
2.10 Set Forming Expressions . . . 18
2.10.1 Enumerated Set Formers . . . 19
2.10.2 Arithmetic Set Formers . . . 20
2.10.3 General Set Formers . . . 20
2.11 Tuple Forming Expressions . . . 22
2.12 Operator Precedence Rules . . . 22
2.13 Left Hand Sides . . . 22
2.14 Assignments . . . 24
2.15 Compound Operators . . . 24
2.16 Boolean Operations . . . 25
2.17 ? Operator . . . 25
2.18 Deletion Operations . . . 26
2.19 λ Expressions . . . 26
2.20 Quantified Expressions . . . 26
2.21 If Statements . . . 27
2.21.1 If Expressions . . . 27
2.22 Case Statements . . . 27
2.22.1 Case Expressions . . . 28
2.23 While and Until Loops . . . 28
2.23.1 Exit Statement . . . 29
2.23.2 Continue Statement . . . 29
2.24 For Loops . . . 29
2.25 Stop Statement . . . 30

2.26 Assert Statement 30
2.27 Null Statements 30
2.28 Packages 30
2.28.1 Package Specifications 30
2.28.2 Package Bodies 31
2.28.3 Importing a Package 31
2.28.4 Compilation Units 32
2.29 Built-In Procedures 32
2.29.1 Atom Generation Procedure 33
2.29.2 Arithmetic Functions 33
2.29.3 Input - Output Procedures 34
2.29.4 String Handling Procedures 36
2.29.5 Type Finding Procedures 38
2.29.6 Environment Access Procedures 38

3 SETL2 Operation Manual 39
3.1 Introduction 39
3.2 Libraries 39
3.3 Installation 40
3.4 Executing The Compiler 40
3.5 Executing The Interpreter 42
3.6 Environment strings 43
3.7 Acknowledgements 43

A A Random Number Generator 45

B The Stable Assignment Problem 49

C A Five Function Calculator 53

1 An Introduction to SETL2

1.1 Overview

SETL2 is a very high level programming language using notation and data types from the theory of finite sets. It is evolved from SETL, developed at New York University by J. T. Schwartz. The principal differences between SETL2 and SETL are:

- SETL2 allows full block structure of procedures, as in Pascal or Ada. SETL allows only one level of procedures, as in C.
- A SETL2 procedure is a first class object with capabilities similar to procedure pointers in C, but somewhat extended. A SETL procedure may only be called, as in Ada.
- In SETL2 an iterator introduces a block with the bound variables local to that block.
- The facilities for separate compilation in SETL2 borrow from Ada the idea of separate module specifications and implementations. The syntax follows that of Ada, but is less complex. Since SETL2 is weakly typed with no type declarations much of the complexity of Ada packages is unnecessary.
- Several of SETL2's control structures use syntax closer to that of Ada.
- SETL2 does not have SETL's data representation sub-language, macros, or backtracking.

The SETL2 system is written in highly portable ANSI C. At present, implementations are available for six computers and operating systems, with more planned or underway.

In the first section of this document, we will explain in detail some of the areas in which SETL2 differs markedly from SETL. We will assume that the reader is familiar with SETL already. If not, we suggest the reader refer to [SDDS86], which is the definitive reference on SETL.

1.2 Nested Procedures

In SETL procedures may not be nested, there is only a main program and procedures at the top level of that program. SETL2 on the other hand allows procedures to be nested to any depth. The syntax of a procedure definition corresponds closely to that of a program definition, as can be seen in the program in figure 1, which prints a recursive tuple using indentation to show its structure.

Notice that unlike many languages with full block structure, nested procedures in SETL2 appear at the *end* of a procedure, not at the beginning. This makes it more convenient to read the program in a top-down manner, and is more consistent with SETL.

1.2.1 Name Scoping

In SETL, a name declared in the main program is visible in all procedures and may not be declared in any of them. An undeclared name is only visible in the program body, not in any procedures. SETL2 is slightly different. SETL2 does allow declaration of a name declared in an enclosing procedure, and such a declaration hides the previously visible name. Undeclared names are implicitly declared (depending on a compiler option – see page 41) in *every* procedure in which they appear. The only

```
program Nesting;

   Tuple := [1,2,[3,4,[5,6,7],8],9,10];
   Print_Tuple(Tuple);

   procedure Print_Tuple(T);

      Recursively_Print_Tuple(T,0);

      procedure Recursively_Print_Tuple(T,Indent);

         -- print Indent spaces

         print(""+/[" " : i in [1 .. Indent]]);

         -- if T is a tuple, print each element indented three more columns

         if is_tuple(T) then
            for x in T loop
               Recursively_Print_Tuple(x,Indent+3);
            end loop;
         else
            print(T);
         end if;

      end Recursively_Print_Tuple;

   end Print_Tuple;

end Nesting;
```

Figure 1: Nested Procedure Example

way to make a name visible in a contained procedure is to explicitly declare it with a `var` declaration, a `const` declaration, or a procedure definition.

It is possible to access hidden names using the construction `<owner>.<name>`, assuming `<name>` would have been visible if not hidden. So for example in the program in figure 2, the variable `a` in the main program is hidden in both procedure `one` and procedure `two`. Either procedure can access that variable by referring to it as `Hidden.a`, but neither procedure can refer to the other's variable `a` by a similar construction.

1.3 Procedures As First Class Objects

Most SETL2 procedures are first class objects. This means that they have values which can be bound to variables, inserted in aggregate data structures, and passed as parameters as well as simply being called. The value of a procedure is accessed by refering to the name of the procedure only, without an argument list. For example, the statement

```
x := time;
```

```
program Hidden;

   var a;
   ...
   procedure one;
      var a;
      ...
   end one;

   procedure two;
      var a;
      ...
   end two;

end Hidden;
```

Figure 2: Hidden Name Example

assigns to `x` the *procedure* `time`, while the statement

```
y := time();
```

calls the procedure `time` and assigns the returned value to `y`. The procedure assigned to `x` above is then called just as a procedure constant, i.e. with the expression `x()`.

1.3.1 Write Parameter Restriction

We started this section with the statement that most procedures are first class objects. More specifically, any procedure whose formal parameters are all read-only are first class objects, those with any read-write or write-only parameters do not have an assignable value. To understand the reason for this restriction, imagine that we did not have the restriction and consider the statement:

```
Lhs := f([x,-,y]);
```

This statement is perfectly legal if `f` is a procedure with a single write parameter. The problem is, what if `f` is a variable which happens to be bound to a procedure? It seems we must check this at run time unless the compiler is able to determine that `f` is indeed a procedure with a single write parameter. In general the compiler can not make such a determination, so the small restriction described above is imposed rather than paying the performance penalty required to check such things at run-time.

1.3.2 Closures

There is a potential ambiguity when nested procedures are passed outside of the scope in which their names are visible. To illustrate, consider the program in figure 3.

In each call to `Plant`, the nested procedure `Seedling` is returned to the main program. When that procedure is called via `First_Crop()`, the enclosing procedure, `Plant`, is no longer active. Since

```
program Farm;

   First_Crop := Plant("corn");
   Second_Crop := Plant("oats");

   print(First_Crop());
   print(Second_Crop());

   procedure Plant(Seed);

      return Seedling;

      procedure Seedling;
         return Seed;
      end Seedling;

   end Plant;

end Farm;
```

Figure 3: Closure Example

`Seedling` returns `Seed`, which is local to `Plant`, what might `First_Crop()` return? There seem to be three possiblilites:

1. It might return Ω, since there may be no binding for `Seed` when `Plant` is not active.
2. It might return `"oats"`, since that was the last valid binding for `Seed`.
3. It might return `"corn"`, since that was the binding for `Seed` at the time `Seedling` was copied.

In fact, the correct answer is 3. When a procedure value is used, SETL2 includes the *closure* of that procedure, or the current activations of enclosing procedures *at that point*. When the procedure is later called, those activations will be reinstated (assuming they are not already active) before the procedure executes. So the farm program above will output:

```
corn
oats
```

A much more detailed explanation of closures may be found in [Mac87].

1.4 Anonymous Procedures – λ Expressions

Conceptually, a λ expression is an expression which yields a procedure where the procedure is defined in the λ expression. As an example, consider the program in figure 4, which just prints the squares of the integers from 1 to 10 using a more general procedure, `Power`.

The λ expression is very similar to a standard procedure definition. It consists of a header, declarations, a statement list, nested procedures, and a tail. A procedure will be declared in the

```
program Curry;

   Square := lambda(x);
               return Power(x,2);
             end lambda;

   for i in [1 .. 10] loop
      print(Square(i));
   end loop;

   procedure Power(x,y);
      return 1 */[x : i in [1 .. y]];
   end Power;

end Curry;
```

Figure 4: λ Expression Example

same scope of the expression with a compiler-generated name. Semantically, the program in figure 4 is identical to the program in figure 5, if the name of the procedure `temporary` were a compiler-generated temporary name.

```
program Curry;

   Square := temporary;

   for i in 1 .. 10 loop
      print(Square(i));
   end loop;

   procedure Power(x,y);
      return 1 */[x : i in [1 .. y]];
   end Power;

   procedure temporary(x);
      return Power(x,2);
   end temporary;

end Curry;
```

Figure 5: Alternative Curry Program

The λ expression is compiled by the SETL2 compiler, not interpreted as a character string. It is less powerful than a λ expression in LISP, or some other interpreted languages for that reason. It is merely a syntactic convenience.

1.5 Separate Compilation – Packages

The SETL system uses modules to implement separate compilation, but SETL2 uses a subset of the Ada package system for that purpose. The general idea is the same, but packages make it much more convenient to import separately compiled modules.

A SETL2 package is normally in two separate compilation units: the package specification and the package body. A package specification contains the names of constants, variables, and procedures which will be visible in any units importing the package, but it will not contain the bodies of any procedures in the package. To illustrate, look at the classic stack package in figure 6. It uses tuples to implement stacks, providing the normal push and pop functions.

```
-- package specification

package Stack_Module;

   procedure Push(rw Stack,Item);
   procedure Pop(rw Stack);

end Stack_Module;

-- package body

package body Stack_Module;

   procedure Push(rw Stack,Item);
      Stack with:= Item;
   end Push;

   procedure Pop(rw Stack);
      if #Stack = 0 then
         return om;
      else
         return Item frome Stack;
      end if;
   end Pop;

end Stack_Module;
```

Figure 6: Package Example

The package specification includes the headers of the two procedures visible to importing units. It could also have contained variable declarations and constant declarations if there were any which should be visible in those units. A program or other package may import a package as follows:

```
program something;
   use Stack_Module;

   ...

end something;
```

The `use` clause imports all the names in the package specification into the program. It must immediately follow the program or package header, before any `const` or `var` declarations or statements.

A potential ambiguity occurs when two packages contain a common name, and a program imports both of those packages. SETL2 adopted Ada rules to handle that occurrence: the names will hide each other. The values bound to those names are still accessible using the construction `<package name>.<name>`

1.6 Bound Variables Are Local To Iterators

In SETL, the scope of a bound variable in an iterator is the same as any other name. So after executing the instruction:

```
y := {x in S | x > 10};
```

the value of `x` would be Ω. It would have been set to each element of `S` during the iteration which built the set, yielding Ω when all elements had been exhausted. In SETL2 the variable `x` would have been locally declared in the iterator, so after the same statement `x` would have the same value it had before the statement.

This characteristic has implications for the `for` loop and quantifier expressions as well. Consider the following loop:

```
for i in [1 .. 10], [x,-,y] in S loop
   ...
   if ... then exit; end if;
   ...
end loop;
```

The bound variables `i`, `x`, and `y` are visible only within the `for` loop. If there are other variables with the same name outside that loop, they will be hidden within the `for`. This means that there is no way to determine whether or not the `exit` statement was taken. There are two ways to handle this situation. First, we could explicitly set shadow variables for the bound variables and test them when the loop terminates. Second, we could use the *value* of the `for` loop as follows:

```
i := for i in [1 .. 10], [x,-,y] in S loop
   ...
   if ... then exit i; end if;
   ...
end loop;
```

Now the `exit` statement returns a value, the bound variable `i`. This value is returned as the value of the `for` loop and assigned to another variable `i` which is *outside* the iterator, and so will be available at the termination of the loop. If no exit is taken, or an exit without a return value is taken, the value of a `for` loop is Ω.

The only exception to the rule that bound variables are local to iterators is in the `exists` expression. That expressions does set its bound variables on exit, to the value found if successful or Ω if unsuccessful.

1.7 Ada-Like Syntax

Many of the control structures in SETL2 differ from those in SETL in that the syntax is more Ada-like. In particular, the `for` loops, `while` loops and `case` statements and expressions are more Ada-like. There is no strong justification for this, we simply believe this syntax is more aesthetically appealing. See the next section for the specific syntax. It is fairly straightforward.

1.8 Summary

SETL2 is a somewhat modernized version of SETL, with syntax and name scoping moved slightly toward those of Ada. It remains a powerful, high-level language with a rich set of built-in data structures and robust operators.

2 SETL2 Reference Manual

2.1 Introduction

This section describes in more detail the syntax and semantics of the SETL2 programming language. We will try to avoid operation details and implementation dependencies as much as possible, and defer that to the next section of this document. There are relatively few such dependencies so the various implementations are very compatible.

A word of advice to the reader: It is generally difficult to learn a programming language from a document written at this level. If you are not familiar with SETL and are unwilling to read a more detailed book on SETL (such as [SDDS86]), then turn to the sample programs in the appendicies and try to use the manual to help you understand them. In the process you will learn the language with less pain than simply reading the manual.

2.2 Notation and Terminology

The description that follows uses square brackets (`[` and `]`) to enclose optional items, angle brackets (< and >) to denote items to be replaced by variable text, and elipses (...) to indicate that the preceeding item may be repeated any number of times. This is occasionally confusing because the symbols `[`, `]`, <, and > are also valid SETL2 separators and operators. To minimize the confusion we will use `<left bracket>` and `<right bracket>` to denote the SETL2 separators `[` and `]`, and provide examples to make our intention clear.

2.3 Lexical Conventions

There are five classes of lexical tokens: identifiers, numeric literals, string literals, operators, and other separators. Blanks and non-graphic control characters other than an end of file character (if one exists on the operating system being used) are ignored except as token separators.

2.3.1 Comments

Comments begin with a double dash (`--`) and extend to the end of a line. They are ignored completely by the SETL2 compiler.

2.3.2 Identifiers

Identifiers are used as names and as reserved words. They must begin with a letter, and be followed by a sequence of letters, digits, or underscores. Both upper and lower-case letters are permitted, but case is not significant, so for example `word_count`, `WORD_COUNT`, and `Word_Count` all represent the same name. A list of reserved words appears in table 1.

and	arb	assert	body	case
const	continue	domain	else	elseif
end	exit	find	for	forall
from	fromb	frome	if	in
incs	lambda	less	lessf	loop
max	min	mod	not	notin
npow	null	or	otherwise	package
pow	procedure	program	range	rd
return	rw	sel	stop	subset
then	until	use	var	when
while	with	wr		

Table 1: Reserved Words

2.3.3 Numeric Literals

The definition of numeric literals was taken from Ada (see [Ada83] or [Bar82] for a more detailed description).

There are two forms of integer literals: decimal and based. A decimal integer literal is simply a sequence of the digits 0 - 9, with optional underscores in all but the first position. The underscores are ignored, and the value of the literal is just the standard base 10 interpretation of the symbol.

A based integer literal is a sequence of digits, a sharp sign (#), another sequence of digits, and a final sharp sign. The first sequence of digits is considered to be the base (it must be between 2 and 36) and the second is the value of the literal in the given base. Alphabetic characters in either case are used for the digits 10 - 36. As with decimal literals, underscores may appear in all but the first position of a sequence of digits, and will be ignored.

Floating point literals may also be decimal or based. A decimal literal is a sequence of digits, a decimal point, and another sequence of digits with the normal base 10 interpretation. A based floating point literal is a sequence of digits, a sharp sign, a sequence of digits, a decimal point, a sequence of digits, and another sharp sign. The interpretation is analogous to based integer literals. Either form may be followed by an exponent, which is an 'E' or 'e', an optional sign, and a sequence of digits. The exponent is a decimal power of the base.

Some examples of valid numeric literals are:

```
12   0       123_456  16#1a#                    -- value 26 decimal
1.5  1.5e10  1.5E-10  2#1.1111_1111_111#e11     -- value 4095.0 decimal
```

The maximum length of an integer literal is 256 characters. Note that larger integers may be represented internally: the restriction is only on literals, not all integers. The maximum size of a floating point literal is implementation dependent.

2.3.4 String Literals

String literals consist of any sequence of characters surrounded by double quotes. They may not include newlines, end of file characters, or unescaped double quotes. Escape sequences allow characters to be included which are difficult or impossible to include by typing them in directly. The escape sequences recognized by SETL2 and the characters that they stand for are given in table 2.

String literals are limited to 256 characters in length. Any size string may be represented internally, but longer strings must be constructed with the concatenation operator (+). Note that unlike C, a null does not terminate a character string. Nulls are simply characters which may be embedded in strings.

\\	Backslash.
\0	Null (a zero byte).
\n	Newline. Newlines are a single line feed character, but are translated to carriage return - line feed pairs on output on systems where that is conventional.
\r	Carriage Return.
\f	Form feed.
\t	Tab.
\"	Double quote. Note that double quotes embedded in string literals must be escaped.
\x*dd*	Hexadecimal code. Here *d* must be a hexadecimal digit 0 .. 9, A .. F in either upper or lower case.

Table 2: String Escape Sequences

2.3.5 Operators

Although some keywords are operators, most operators are made up of one or more graphic characters other than letters and digits. A complete list of non-keyword operators is:

```
+       -       *       /       **
:=      #       ?       =       /=
<       <=      >       >=
```

2.3.6 Separators

Separators are also made up of graphic characters other than letters and digits, but do not denote any operation to be performed. A complete list of separators is:

```
;       ,       :       (       )
{       }       [       ]       .
..      |       =>
```

2.4 Overall Program Structure

The overall structure of a SETL2 program is as follows:

```
program <program name> ;

   <use section>
   <constant and variable declaration section>
```

```
        <program body>
        <procedure section>

    end [ <program name> ] ;
```

The `<use section>` is a sequence of clauses of the form `use package name`. See 2.28.3 for more details. The `<constant and variable declaration section>` defines names which will be visible throughout the program. The `<program body>` is a list of statements making up the main procedure of the program. And finally `<procedures>` is a list of procedures visible at the top level of the program. An unnecessarily complex SETL2 *hello world* program showing this overall structure is:

```
program hello_world;

   var hello,
       world;

   hello := return_hello();
   world := return_world();

   print(hello," ",world);

   procedure return_hello;
      return "hello";
   end return_hello;

   procedure return_world;
      return "world";
   end return_world;

end hello_world;
```

2.5 Declarations and Scope of Names

There are two kinds of data declarations in SETL2: variable declarations and constant declarations. The syntax of a variable declaration is as follows:

```
var <v1> [ := <expression 1> ]
    [, <v2> [ := <expression 2> ] ] ... ;
```

The names `<v1>`, `<v2>` ... will be declared as variables in the current unit (a program, package, or procedure). Any variables with associated value expressions will be initialized to the values of those expressions when the unit is activated, exactly as if there were corresponding assignment statements inserted in front of the unit body. Any names declared with a `var` clause will be visible in all nested procedures. This is an important point: SETL2 allows variables to be implicitly declared (depending on a compiler option – see page 41). Variables which are implicitly declared are not visible in nested procedures, those which are explicitly declared are.

The syntax of a constant declaration is as follows:

```
const <c1> := <expression 1>
      [, <c2> := <expression 2> ] ... ;
```

The names <c1>, <c2> ... will be declared as constants in the current unit. They may be used in expressions but not in left hand side contexts. The associated value expressions must only contain literals and previously defined constants as operands, although most of the operators are valid. They may not use constants in imported packages, only constants within the current compilation unit. Constants are visible in all nested procedures, as well as the unit in which they are declared.

2.5.1 Selectors

Unlike most other programming languages, SETL2 has no record data type. Tuples are generally used in situations where records are used in other languages, but these are somewhat less convenient since the components of a tuple are numbered rather than named. To recapture the ability to name elements of a tuple as fields would be named in a record, SETL2 provides selector declarations. The syntax of a selector declaration is:

```
sel <n1> ( <integer literal> )
      [, <n2> ( <integer literal> ) ] ... ;
```

For example, here is a valid selector declaration:

```
sel first_elem(1),
    second_elem(2),
    third_elem(3);
```

After a selector declaration is made, the parser will translate expressions of the form `.<name>` into `(<integer>)`. So continuing with the example above, the expression `Tuple.first_elem` is translated into `Tuple(1)`. It is important to note that this translation is not restricted to tuples. Selectors can also be used to reference string and map values, and even to call procedures. They are overloaded to the extent parentheses are overloaded.

2.6 Statement Lists

Statement lists are used as program and procedure bodies as well as clauses in some kinds of statements. A statement list is just a sequence of statements with each statement in the list followed by a semicolon. Notice that semicolons are used as *terminators* in SETL2, not as separators as in Pascal.

2.7 Procedures

A *procedure* is a named unit containing local data and statements which may be executed by calling the procedure. The syntax of a procedure definition is:

```
procedure <procedure name> [ ( [ <formal list> ] ) ] ;

   <constant and variable declaration section>
   <procedure body>
   <procedure section>

end [ <procedure name> ] ;
```

where the `<formal list>` is:

```
[ <mode 1> ] <formal 1> [, [ <mode 2> ] <formal 2> ] ...
```

The `<mode>`'s and their meanings are:

`rd` or *empty*	read-only parameters
`rw`	read-write parameters
`wr`	write-only parameters

As an example in the following discussion look at the procedure below.

```
procedure doverylittle(a, rd b, rw c, wr d);

   var e, f;

   d := c * a;
   c := b;

end doverylittle;
```

The first parameter, `a`, has no mode and the second parameter, `b` is declared `rd`. These are read-only parameters. Any callers of `doverylittle` must provide values for these parameters, and within the body of `doverylittle` these are constant.

The third parameter, `c`, is declared `rw`. The actual parameter the caller provides must be valid both in left and right hand side contexts. The value provided will be copied into `c` on entry to `doverylittle`. The procedure is free to modify that parameter just as any other variable, and on return the value of `c` will be copied into the caller's actual parameter.

The fourth parameter, `d` is declared `wr`. The caller must use a valid left hand side for this parameter. On entry to the procedure, `d` will be initialized to Ω. The procedure is free to modify that parameter just as any other value, and on return the value of `d` will be copied into the caller's actual parameter.

Note that in SETL2, parameters are transfered by *copying*, not by reference.

Procedures are called by giving the name of the procedure followed by a parenthesized list of actual arguments. For example, a call to the procedure above might be

```
doverylittle(1,2,x,[a,-,b]);
```

Notice that we could not have used a literal as the third or fourth actual arguments, and that the fourth argument need not have a right hand side value.

Procedures with no formal parameters are called with empty parentheses (for example: `f()`) in right hand side contexts. When used as statements however, the parentheses are optional.

2.7.1 Return Statements

Within the body of a procedure there may appear one or more `return` statements, which cause an immediate termination of the procedure and a return to the caller. The syntax of a return statement is as follows:

```
return [ <expression> ]
```

The optional `<expression>` is returned to the caller as the value of the procedure call. If the procedure call was a statement, this return value is discarded. If the procedure call was an expression and no return value is given, Ω will be returned. If no `return` statement is encountered and control reaches the end of the procedure, Ω will be returned.

2.8 Basic Data Types

There are six basic data types in SETL2: integers, floating point numbers, character strings, atoms, procedures, and an undefined value.

2.8.1 Integers

SETL2 provides support for integers, just as most programming languages do, but with one important difference: in SETL2 the size of an integer is limited only by available memory, which for practical purposes is infinite. Table 3 lists the valid operations on integers.

`-`i	Yields the negative of i.
i `+` j	Yields the sum of i and j.
i `-` j	Yields the difference of i and j.
i `*` j	Yields the product of i and j.
i `/` j	Yields the integer part of the quotient of i by j. An error results if j is zero.
i `**` j	Yields i to the jth power. An error results if j is negative.
i `mod` j	Yields the remainder of i divided by j. An error results if j is zero.
i `max` j	Yields the larger of i and j.
i `min` j	Yields the smaller of i and j.
i `=` j	Yields `true` if i and j are the same, `false` otherwise.
i `/=` j	Yields `true` if i and j are different, `false` otherwise.
$i > j$	Yields `true` if i is greater than j, `false` otherwise.
$i < j$	Same as $j > i$.
$i >= j$	Yields `true` if i is no smaller than j, `false` otherwise.
$i <= j$	Same as $j >= i$.

Table 3: Integer Operators

2.8.2 Floating Point Numbers

SETL2 supports floating point numbers with the usual limitations. The floating point representation varies with the implementation, but is always just an approximation to a floating point number. Rounding errors apply, so for example `(x / 100.0) * 100.0` does not necessarily equal `x`. Table 4 lists the valid operations on floating point numbers.

`-`*i*	Yields the negative of *i*.
i `+` *j*	Yields the sum of *i* and *j*.
i `-` *j*	Yields the difference of *i* and *j*.
i `*` *j*	Yields the product of *i* and *j*.
i `/` *j*	Yields the quotient of *i* by *j*. An error results if *j* is zero.
i `**` *j*	Yields *i* to the *j*th power. An error results if *j* is negative.
i `max` *j*	Yields the larger of *i* and *j*.
i `min` *j*	Yields the smaller of *i* and *j*.
i `=` *j*	Yields `true` if *i* and *j* are the same, `false` otherwise.
i `/=` *j*	Yields `true` if *i* and *j* are different, `false` otherwise.
i > *j*	Yields `true` if *i* is greater than *j*, `false` otherwise.
i < *j*	Same as *j* > *i*.
i >= *j*	Yields `true` if *i* is no smaller than *j*, `false` otherwise.
i <= *j*	Same as *j* >= *i*.

Table 4: Floating Point Operators

2.8.3 Strings

A string is simply a sequence of characters. The length of a string is limited only by available memory, which for practical purposes is infinite. Table 5 lists the valid operations on strings.

2.8.4 Atoms

Atoms are generated values used primarily as one might use a pointer in other languages. They are generated by calling the built-in procedure `newat`, which returns a new unique atom value with each call. They may then be used as unique keys in maps. The only operations valid on atoms are assignment and equality tests.

2.8.5 Procedures

Procedures *all of whose formal parameters are read-only* are first class objects in SETL2. They can be passed as parameters, assigned to variables, and stored in sets, maps, or tuples as well as being executed. They can not be written to files and read back in during a different execution, the arithmetic operations will fail if they are used as operands, and the text of a procedure can not be changed. Generally, if you consider atoms to be first class objects then procedures are first class objects. Similar restrictions apply.

2.8.6 Ω - Undefined Value

SETL2 has a constant, `om` (Ω), which stands for the undefined value. It is generally used as the result of failing operations. Most operations cause an abnormal program end if Ω is used as an operand.

`#`*s*	Yields the length of *s*.
s(*i*)	Yields the *i*th character of the string *s*. If *i* is less than or equal to zero, an error results and the program is terminated. If *i* is greater than the length of the string the expression yields Ω.
s(*i* .. *j*)	Yields the *slice* of *s* from the *i*th to the *j*th character. If $i = j + 1$ then a null string is returned. If *i* is less than or equal to zero, $i > j + 1$, or *j* is greater than the length of *s* an error results and the program is terminated.
s(*i* ..)	Yields the *slice* of *s* from the *i*th character to the end of the string. If *i* is one more than the length of *s* then a null string is returned. If *i* is less than or equal to zero or *i* is greater than the length of *s* plus one an error results and the program is terminated.
s `+` *ss*	Concatenates the strings *s* and *ss*.
s `*` *i*	Concatenates *i* successive copies of *s*. The `*` operation is commutative.
s `=` *ss*	Yields `true` if *s* and *ss* are the same, `false` otherwise.
s `/=` *ss*	Yields `true` if *s* and *ss* are different, `false` otherwise.
s > *ss*	Yields `true` if *s* is greater than *ss*, `false` otherwise. This and the other ordering tests are dependent upon the particular character set used by the system. In the current implementations that is ASCII, but it should not be assumed that all future implementations will use ASCII.
s < *ss*	Same as *ss* > *s*.
s >= *ss*	Yields `true` if *s* is no smaller than *ss*, `false` otherwise.
s <= *ss*	Same as *ss* >= *s*.
s `in` *ss*	Yields `true` if *s* is a substring of *ss*, `false` otherwise.
s `notin` *ss*	Yields `false` if *s* is a substring of *ss*, `true` otherwise.

Table 5: String Operators

2.9 Compound Data Types

There are two compound data types in SETL2: sets and tuples. SETL2 also supports special operations on maps, which is a subset of the set data type.

In this section we will describe the compound types from a high level, defering to 2.10 a discussion of the ways that they may be created.

2.9.1 Sets

A set in SETL2 closely matches the mathematical idea of a finite set. It is simply an unordered collection of distinct SETL2 values. Table 6 lists the valid operations on sets.

2.9.2 Tuples

A tuple is a sequence of SETL2 values, indexed from 1 to ∞. The length of a tuple is limited only by available memory, which for practical purposes is infinite. We always think of tuples as having infinite length. It is always possible to assign or refer to elements past the internal length of the

`#`*s*	Yields the cardinality of the set *s*.
`arb` *s*	Yields an arbitrarily selected element of the set *s*.
`pow` *s*	Yields the *power* set of set *s*. Be a bit careful with this operator. There are a number of algorithms which can be elegantly expressed using the `pow` operator, but they can be disastrously expensive to execute unless the source set is very small.
s `+` *ss*	Yields the union of the sets *s* and *ss*.
s `-` *ss*	Yields the difference of the sets *s* and *ss*, i.e. the set of all elements of *s* which are not elements of *ss*.
s `*` *ss*	Yields the intersection of the sets *s* and *ss*.
s `mod` *ss*	Yields the symmetric difference of two sets *s* and *ss*, i.e. the set of all elements which are in *s* or *ss*, but not in both.
s `npow` *i*	Yields the set of all subsets of *s* which contain exactly *i* elements. If *k* is negative an error results and the program is terminated. The `npow` operator is commutative.
s `with` *v*	Yields the set $s \bigcup \{v\}$.
s `less` *v*	Yields the set $s - \{v\}$.
v `in` *s*	Yields `true` if $v \in s$, `false` otherwise.
v `notin` *s*	Yields `false` if $v \in s$, `true` otherwise.
s `=` *ss*	Yields `true` if *s* and *ss* are the same, `false` otherwise.
s `/=` *ss*	Yields `true` if *s* and *ss* are different, `false` otherwise.
s `subset` *ss*	Yields `true` if $s \subseteq ss$, `false` otherwise.
s `incs` *ss*	Yields `true` if $s \supseteq ss$, `false` otherwise.

Table 6: Set Operators

tuple. Those elements simply contain Ω. Table 7 lists the valid operations on tuples.

2.9.3 Maps

Conceptually, maps in SETL2 are similar to the mathematical notion of maps, i.e. a set of ordered pairs. They are implemented in the language as sets of tuples, with each tuple having exactly two elements. Notice that from the programmer's perspective there is not a dedicated type, *map*. Rather, maps are just special kinds of sets. Table 8 lists the specific operations on maps, but keep in mind that since a map is just a special kind of set all of the set operations are available as well.

2.10 Set Forming Expressions

Sets are created in SETL2 with set forming expressions, which describe the elements of the set. There are three basic types of these: enumerated set formers, arithmetic set formers, and general set formers.

`#`*t*	Yields the length of *t*. In a sense tuples really have infinite length, this is the index of the last non-Ω element.
t(*i*)	Yields the *i*th element of the tuple *t*. If *i* is less than or equal to zero, an error results and the program is terminated. If *i* is greater than the length of the tuple the expression yields Ω.
t(*i* .. *j*)	Yields the *slice* of *t* from the *i*th to the *j*th element. If $i = j + 1$ then a null tuple is returned. If *i* is less than or equal to zero, $i > j + 1$, or *j* is greater than the length of *t* an error results and the program is terminated.
t(*i* ..)	Yields the *slice* of *t* from the *i*th element to the end of the tuple. If *i* is one more than the length of *t* then a null tuple is returned. If *i* is less than or equal to zero or *i* is greater than the length of *t* plus one an error results and the program is terminated.
t `+` *tt*	Concatenates the tuples *t* and *tt*.
t `*` *i*	Concatenates *i* successive copies of *t*. The `*` operation is commutative.
t `with` *v*	Appends the element *v* to the tuple *t*.
t `=` *tt*	Yields `true` if *t* and *tt* are the same, `false` otherwise.
t `/=` *tt*	Yields `true` if *t* and *tt* are different, `false` otherwise.
v `in` *t*	Yields `true` if *v* is one of the elements of *t*, `false` otherwise.
v `notin` *t*	Yields `false` if *v* is one of the elements of *t*, `true` otherwise.

Table 7: Tuple Operators

2.10.1 Enumerated Set Formers

The simplest kind of set former is one in which we just enumerate all of the elements in a set. The syntax is

```
{ <v1> [, <v2>] ... }
```

Some examples of enumerated set formers are:

```
{1,2,3}        {"a",1,3.0}        {1,{1,{1,2},2},2,{3}}
```

`domain` *m*	Yields the set of all the left elements of the pairs in *m*.
`range` *m*	Yields the set of all the right elements of the pairs in *m*.
m(*v*)	If there is exactly one pair in *m* with the value *v* as the left element then *m*(*v*) yields the right element of that pair. If there is no such pair or more than one the value is Ω.
m(*v1*,*v2*,...*vn*)	Same as *m*([*v1*,*v2*,...*vn*])
m{*v*}	Yields the set of values, *y*, such that the pair [v,y] is in the map *m*. *m*{*v*} is called the image set of *m* at the point *v*.
m{*v1*,*v2*,...*vn*}	Same as *m*{[*v1*,*v2*,...*vn*]}

Table 8: Map Operators

Notice that sets need not be homogeneous (elements may be of different types). Also notice that sets may contain embedded sets or tuples, as in the third example above.

2.10.2 Arithmetic Set Formers

An arithmetic set former creates a set of integers from two endpoints and an intermediate point which defines an increment. The syntax is

```
{ <v1> [, <v2>] .. <v3> }
```

The set created will contain the integers `<v1>`, `<v2>`, `<v2> + (<v2> - <v1>)` ... `<v3>`. The middle element is optional and defaults to `<v1> + 1`.

Some examples of enumerated set formers are:

```
{1 .. 10}      {5,4 .. 1}
```

The first set above is the set of integers from 1 to 10 and the second is the set of integers from 1 to 5.

2.10.3 General Set Formers

A general set former has the form

```
{ [ <expression> : ] <iterator> [ | <condition> ] }
```

The `<iterator>` produces values from one or more sources, the `<condition>` weeds out values which should not be included in the set, and `<expression>` is a function of the values which should be inserted in the set. This is somewhat complex to explain, so let's start with a fairly full example and see how it works. Look at the expression

```
{x**y : x in S1, y in S2 | x < 5 and y > 2}
```

Assume `S1` and `S2` are sets. Then in the iteration part, each element of `S1` will be produced in `x` and each element of `S2` will be produced in `y`. If `S1` and `S2` have the values `{3,4,5}` and `{2,3,4}` respectively, then the following pairs of values will be produced:

```
x = 3, y = 2        x = 3, y = 3        x = 3, y = 4
x = 4, y = 2        x = 4, y = 3        x = 4, y = 4
x = 5, y = 2        x = 5, y = 3        x = 5, y = 4
```

The `<condition>` part of the set former determines which of these values will be kept, so continuing with the example we have

```
x = 3, y = 3        x = 3, y = 4
x = 4, y = 3        x = 4, y = 4
```

Finally, the `<expression>` part of the set former determines the values which will actually be inserted in the set, which in this case is:

```
{3**3, 3**4, 4**3, 4**4}
```

Now let's look in a bit more detail at `<iterators>`, since they are the most complex component of a set former. An iterator has the general form:

```
<iterator spec> [, <iterator spec>] ...
```

where an `<iterator spec>` has one of the two following forms:

```
<lhs1> in <source1>

<lhs2> = <source2> ( <lhs3> )
```

In the first form, `source1` may be either a set, a tuple, or a string. If a set, then each element of the set will be assigned to `<lhs1>` in an arbitrary order. If a tuple, then each element of the tuple will be assigned to `<lhs1>` in the order in which they appear in the tuple. If a string, then each character of the string will be assigned to `<lhs1>` from the left of the string to the right.

In the second form, `<source2>` must be a map, a tuple, or a string. If a map, the iterator will produce each pair in the map, assigning the left element of the pair to `<lhs3>` and the right element to `<lhs2>`. If a tuple or string, `<lhs3>` will be assigned the integers `1` to `#<source2>` and `<lhs2>` will be assigned the corresponding elements of the source.

Each of `<lhs1>`, `<lhs2>`, and `<lhs3>` may have two distinct forms:

```
<name>

<left bracket> <lhs1> [, <lhs2> ] ... <right bracket>
```

In the first case the simple variable `<name>` will be assigned the successive elements of the source. In the second, each element of the source must be a tuple. Then `<lhs1>`, `<lhs2>`, ... are set to the corresponding elements of that tuple. Any of `<lhs1>`, `<lhs2>` ... may be `-`, in which case the corresponding tuple element is skipped. An example of the second form is

```
[a,-,[b,c],d]
```

In this example, the source should produce tuples four elements in length. The third element of each tuple should be a tuple of length two, the others may be anything.

We refer to the simple variables in the left hand sides as *bound variables*. They are repeatedly bound to values from the source. These variables are only visible within the set former, and hide any variables outside the set former with the same names.

Now we can return to the other components of set forming expressions, namely `<expression>` and `<condition>`. The `<condition>` is any SETL2 expression which yields a boolean value. All values produced by the iterator which cause the condition to yield `true` will be kept. The `<condition>`

may be omitted only if the initial `<expression>` *is* used, since otherwise we can not distinguish a general set former from an enumerated set former.

The `<expression>` is any SETL2 expression, and defines the actual values inserted in the set. It may be omitted only if two conditions are satisfied:

1. The `<condition>` is used. Otherwise we can not distinguish a general set former from an enumerated set former.
2. The `<iterator>` is of the form `<lhs> in <source>`. If there is than one iterator clause, or if a map iterator is used it is necessary to include an `<expression>` to define what value must actually be placed in the set.

If `<expression>` is omitted then the left hand side of the iterator is inserted in the set.

2.11 Tuple Forming Expressions

A tuple forming expression is nearly identical to a set forming expression. The only difference is that the outer braces are replaced by square brackets. Each value produced will be placed in the created tuple in sequence. Duplicates will not be discarded, as they would in a set former. Some examples of tuple forming expressions are:

```
[x**y : x in S1, y in S2 | x < 5 and y > 2]
[x in S | x /= om]
```

2.12 Operator Precedence Rules

We have listed a variety of operators with the types upon which they are defined, but so far not given any information about how they may be combined in expressions. Generally, the operators may be combined as in SETL, Pascal, or other programming languages. Table 9 lists all the operator precedences. We have fewer precedence levels than SETL, but even so there are probably too many levels to remember easily. We strongly encourage the use of parentheses to clarify the intent, even when those parentheses are not required by the language.

2.13 Left Hand Sides

Left hand sides get their name from the fact that they are the expressions which may appear on the left of an assignment symbol. They are somewhat more general than that however, they are really the set of expressions to which we can assign a value in any context. We will refer to this value as the right hand side value for now. In SETL2 left hand sides take a variety of forms. The simplest form is just a variable name. If this form is used then the variable is assigned the right hand side value directly. The second form of left hand side is:

```
<lhs> ( <expr1> )
```

In this form the value of `<lhs>` must be a map, a tuple, or a string. If a map, the value of the map at `<expr1>` is set to the right hand side value. If a tuple or string, then the value of `<expr1>` must be

8	:= (on left side), assignment operators on left side.
7	FROM FROMB FROME
6	All unary operators.
5	** (**Note:** associates to the right, not left.)
4	* / MOD ?
3	+ - MAX MIN
2	= /= < <= > >= in notin subset incs
1	AND OR (**Note:** AND and OR do not associate. Parentheses must be given when these are mixed in an expression.)
0	:= (on right side), assignment operators on right side.

Table 9: Operator Precedences

an integer, and the corresponding element of the tuple or string is set to the right hand side value. A similar construction which only works on maps is

```
<lhs> ( <expr1> , <expr2> [ , <expr3> ] ... )
```

This is semantically equivalent to

```
<lhs> ( <left brace> <expr1> , <expr2> [ , <expr3> ] ... <right brace> )
```

That is, we make a tuple of the arguments and use that as the key to the map. The next form is

```
<lhs> ( <expr1> .. <expr2> )
```

In this form the value of <lhs> must be a tuple or a string, both <expr1> and <expr2> must be integers, and the right hand side value must have the same form as <lhs>. A *slice* of <lhs> from <expr1> to <expr2> is set to the right hand side value. A variant of the slice assignment is the tail assignment, which has this syntax:

```
<lhs> ( <expr1> .. )
```

In this form the value of <lhs> must be a tuple or a string, <expr1> must be an integer, and the right hand side value must have the same form as <lhs>. A *slice* of <lhs> from <expr1> to the end is set to the right hand side value. Another assignment to maps is the image set assignment, which has the following syntax:

```
<lhs> { <expr1> }
```

In this form the value of <lhs> must be a map and the right hand side value must be a set. Any pairs with <expr1> as the left hand element are first removed from the map. Then pairs with <expr1> on the left and elements from the right hand side value are created and added to the map. Like the other form of map assignment, we may have a list of expressions inside the braces, which will be made into a tuple. The last form of left hand side is used to disassemble tuples. It has the syntax:

```
<left bracket> <lhs1> [ <lhs2> ... ] <right bracket>
```

In this form the right hand side value must be a tuple. Then `<lhs1>`, `<lhs2>`, ... are set to the corresponding elements of that tuple. Any of `<lhs1>`, `<lhs2>` ... may be `-`, in which case the corresponding tuple element is skipped. For example, in the expression

```
[a,b,-,c] := [1,2,3,4]
```

`a` is assigned `1`, `b` is assigned `2`, and `c` is assigned `4`.

As mentioned above, the most common use of left hand side expressions is in assignments, but they are also used as actual parameters to procedures with write-only formal parameters.

2.14 Assignments

The general form of an assignment is

```
<left hand side> := <expression>
```

where `<expression>` is some value-yielding expression, and `<left hand side>` is one of the forms described above. Any form of assignment may be used either as a statement or as an expression. If used as an expression the value yielded is the value of the expression on the *right* hand side of the assignment symbol, not the left as in some other languages.

There is also an assignment form of binary operator, which looks like

```
<left hand side> <binary operator> := <expression>
```

For example, `x +:= y;`. Semantically, expressions of this form are identical to

```
<left hand side> := <left hand side> <binary operator> <expression>
```

This is merely a syntactic convenience.

2.15 Compound Operators

A *compound* operator is used to combine the elements of a set or tuple using a binary operator. The syntax of a compound operator is

```
<binary operator> / <source>
```

The value of this expression is the value of the expression formed by listing the elements of the source and placing the binary operator between each pair of elements, For example,

```
+/{1,2,3,4,5}   =  1+2+3+4+5 = 15
```

Note that if the source is a set, the elements will appear in an unspecified order, but if the source is a tuple they will appear from left to right. Any binary operator may be used in a compound operator, but they are most useful in summing or finding a largest or smallest value. If the source operand contains no elements, the value of a compound operation is Ω. If it contains only one element then the value of the operation is that element.

There is also a set of binary compound operators with the syntax

```
<first> <binary operator> / <source>
```

which is semantically equivalent to

```
<first> <binary operator> (<binary operator> / <source>)
```

It merely provides a first element for the compound operator.

2.16 Boolean Operations

SETL2 has three boolean operators, `and`, `or`, and `not`. The `and` operator is an infix binary operator which always evaluates its left operand, evaluates its right operand if the left operand is `true`, and yields the logical conjunction of the two operands. Similarly, the `or` operator is an infix binary operator which always evaluates its left operand, evaluates its right operand if the left operand is `false`, and yields the logical disjunction of the two operands. Note that each of these operators will not evaluate their right operands if not necessary to determine the result of the operation. If the evaluation of the right operand has a side effect, that side effect might not occur.

The `and` and `or` operators do not associate without parentheses controlling the order of evaluation. So for example, the expression

```
a = b and c = d or e = f
```

will produce a syntax error rather than automatically associating as

```
(a = b and c = d) or e = f
```

as some languages will allow.

The `not` operator simply produces the logical negation of its operand.

2.17 `?` Operator

The question mark operator allows an economical expression of the most common form of conditional assignment. It evaluates its left operand, and if that is not Ω yields it as the result. Otherwise it evaluates the right operand and yields that result. So the following two expressions assigne the same value to `a`:

```
a := x?y;
a := if (t := x) /= om then t else y end if;
```

2.18 Deletion Operations

There are three *deletion* operators, `from`, `fromb`, and `frome`. They are used as follows:

```
<lhs1> from <lhs2>
```

Here `<lhs1>` can be any left hand side value, but `<lhs2>` must be valid in both left and right hand side contexts. In particular, the tuple disassembly form is not valid. The `from` operator deletes an unspecified element from `<lhs2>` and assigns that element to `<lhs1>`. The value of `<lhs2>` must be a set, tuple, or string.

The `fromb` and `frome` operators perform a similar operation on tuples and strings. `fromb` takes the first item in the right hand side value and `frome` takes the last.

The `from` expressions may be used either as statements or expressions. If used as expressions they yield the value extracted from the right hand side.

2.19 λ Expressions

Conceptually, a λ expression is an expression which yields a procedure where the procedure is defined in the λ expression. The syntax of a λ expression is:

```
lambda [( [<p1> [, <p2>] ...] )];
   <procedure body>
end lambda
```

The parameter list `[( [<p1> [, <p2>] ...] )]` is similar to the parameter list in a procedure header, except that modes are not allowed. All parameters are read-only. The `<procedure body>` may include local declarations, a statement list, and nested procedures, just as a normal procedure can. The names visible within the procedure are those visible at the point of the expression. The procedure defined has no name, so must be used in a value-yielding context. For example, the statement

```
f := lambda(x);
        return p(x,5);
     end lambda;
```

assigns to `f` an unnamed procedure which takes a single argument. The procedure may then be called with an expression like `f(a)`.

2.20 Quantified Expressions

There are two quantified expressions: `exists` and `forall`. The `exists` expression has the following syntax:

```
exists <iterator> | <condition>
```

The `<iterator>` is the same as `<iterator>` in set formers (see 2.10). Each of the values produced by the iterator is checked to see whether `<condition>` is `true`. If so, the iteration stops and the value of the `exists` expression is `true`. The bound variables of the iterator will remain set to the values found. If no such set of values is found, the value of the `exists` expression is `false` and the values of the bound variables will be Ω.

The syntax of a `forall` expression is as follows:

```
forall <iterator> | <condition>
```

This is semantically equivalent to

```
not(exists <iterator> | not(<condition>) )
```

with the exception that bound variables in a `forall` expression are local to the expression. They do not remain set on exit.

2.21 If Statements

The syntax of an `if` statement is:

```
if <expr1> then
   <statement list 1>
[ [ elseif <expr2> then
   <statement list 2> ] ... ]
[ else
   <statement list 3> ]
end if
```

First `<expr1>` is evaluated, which should yield either `true` or `false`. If `true`, then the first statement list is executed. If `false` and there is a following `elseif` clause then that is evaluated as if it were an `if` statement. If there is no following `elseif` clause but there is an `else` clause then `<statement list 3>` is executed.

2.21.1 If Expressions

`if` expressions are nearly identical to `if` statements. The difference is that the statement lists are replaced by single expressions. Note that no semicolons are necessary after the expressions. An example of an `if` expression is:

```
x := if y > 5 then y else z end if;
```

2.22 Case Statements

There are two distinct forms of the case statement. The first is

```
case <expr1>
   when <expr2> [, <expr3>] ... =>
      <statement list 1>
   [ when <expr4> [, <expr5>] ... =>
      <statement list 2> ] ...
   [ otherwise =>
      <statement list 3> ]
end case;
```

This is a multi-way branch, or switch statement as found in most programming languages. First `<expr1>` is evaluated. If that value is the same as `<expr2>` or `<expr3>` ... then `<statement list 1>` is executed. If the value of `<expr1>` is the same as `<expr4>` or `<expr5>` ... then `<statement list 2>` is executed, and so on. If `<expr1>` does not match any `when` clause then `<statement list 3>` is executed.

There is one difference from most languages we would like to point out here: `<expr2>`, `<expr3>`, etc. need not be *constant* expressions. If they happen to be constant expressions the compiler will build a jump table, and execution will be faster. But variable expressions are allowed by the language.

The second form of `case` is

```
case
   when <expr1> =>
      <statement list 1>
   [ when <expr2> =>
      <statement list 2> ] ...
   [ otherwise =>
      <statement list> ]
end case;
```

This is similar to a *guarded* statement. Each of `<expr1>`, `<expr2>`, etc. is evaluated in an unspecified order. When one is found which evaluates to `true`, the associated statement list is executed. If none of the expressions evaluate to `true` then the `otherwise` clause is taken.

Notice that we said that the clauses are evaluated in an *unspecified* order, not a random order. The order can not be controlled by the programmer but is not random.

2.22.1 Case Expressions

As with the `if` statement, there is an expression form of the `case` statement. Just replace each of the statement lists by expressions in the `case` statement. The value returned will be the value of the `when` clause executed.

2.23 While and Until Loops

The syntax of a `while` loop is:

```
while <expression> loop
```

```
   <statement list>
end loop
```

The execution of a `while` loop is just like that in any other programming language: the `<expression>` is evaluated, and if it yields `false` the loop stops. Otherwise the statement list is executed and the loop repeats. There is also an expression form of a `while` loop. See the `exit` statement for a description of the value of `while` loops used in this manner.

The `until` statement and expression is like the `while` statement and expression, except that the test is reversed and at the end of the loop.

2.23.1 Exit Statement

The `exit` statement provides for an abnormal termination of `while`, `until`, and `for` loops. The syntax is:

```
exit [ <expression> ]
```

When encountered, an `exit` immediately breaks out of the loop. If it has an associated `<expression>`, the value of the `<expression>` is yielded as the value of the loop. If there is no `<expression>` or the loop terminates without encountering an `<exit>` then the value of the loop is Ω.

The `exit` statement is illegal except within a loop.

2.23.2 Continue Statement

The `continue` statement causes a branch to the end of a `while`, `until`, or `for` loops. It is frequently used to avoid deeply nested `if` statements within such loops. The syntax is simply

```
continue
```

The `continue` statement is illegal except within a loop.

2.24 For Loops

The syntax of a `for` loop is:

```
for <iterator> [ | <condition> ] loop
   <statement list>
end loop
```

The `<iterator>` is the same as in set forming expressions (see 2.10). Values are produced by the `<iterator>` and screened with the `<condition>` exactly as in a set former. If `<condition>` is omitted then all values will be used. For each set of values produced the statement list is executed. The scope of any bound variables in the `<iterator>` is limited to the loop.

2.25 Stop Statement

The `stop` statement immediately terminates the program. It may be used within the main program or within procedures. It normally is used in a severe error condition, in which the programmer doesn't see any reasonable way to continue.

2.26 Assert Statement

The `assert` is most useful during debugging, when it is used to enforce program invariants. The syntax of an `assert` is

```
assert <expression>
```

The `<expression>` should yield either true or false, and the action taken depends upon a command line option on the interpreter (see 3.5). There are three choices, assertions may be skipped altogether, successful assertions may be skipped but failing assertions stop the program, or successful assertions may be logged with failing assertions stopping the program.

2.27 Null Statements

It is occasionally useful to have an empty statement list, as in the following (illegal) statement

```
if all_is_well then
else print("O my gosh");
end if;
```

The grammar of SETL2 does not allow empty statement lists, so to handle those situations it provides the null statement so the above could (legally) be written as

```
if all_is_well then null;
else print("O my gosh");
end if;
```

2.28 Packages

A *package* is a module at the same level as a program which provides variables, constants, and procedures which may be imported by other packages or programs. There are two components to the source of a package: a package specification and a package body. The package specification describes the names which are visible outside the package, and the package body defines local data and the definitions of the procedures in the package.

2.28.1 Package Specifications

The syntax of a package specification is

```
package <package name> ;

   <variable and constant declarations>
   <procedure declarations>

end [ <package name> ] ;
```

The `<variable and constant declarations>` include `var`, `const`, and `sel` declarations just as in a program. The names declared will be available to any unit importing the package. The procedure declarations are the headers of any procedures in the package. The package specification must be compiled before the associated package body.

2.28.2 Package Bodies

A package body contains data visible throughout the package, but not outside the package, along with the complete definitions of the procedures in the package. The syntax of a package body is

```
package body <package name> ;

   <use section>
   <constant and variable declaration section>
   <procedure section>

end [ <package name> ] ;
```

The `<use section>` is a sequence of clauses of the form `use package name`. See 2.28.3 for more details. The `<constant and variable declaration section>` defines names which will be visible within the package, but not outside the package. And finally the `<procedure section>` is a list of procedure definitions including all procedures listed in the package body, and possibly others visible only within the package. An example illustrating all of these components is in figure 7, and a more useful but less comprehensive example is in Appendix A.

2.28.3 Importing a Package

To import a package a use clause is placed before the declaration section of a program or package body. The syntax of a use clause is

```
use <package name 1> [ , <package name 2> ] ... ;
```

For example, a program importing the package in figure 7 is

```
program Something;

   use Anything;

   visible_proc(1,2);

end Something;
```

```
package Anything;

   var visible_var;
   const visible_const := 5;

   procedure visible_proc(p1,p2);

end Anything;

package Anything;

   var hidden_var;
   const hidden_const := 10;

   procedure visible_proc(p1,p2);
      print("Hello");
   end visible_proc;

   procedure hidden_proc(p1,p2);
      print("Goodbye");
   end hidden_proc;

end Anything;
```

Figure 7: Package Example

2.28.4 Compilation Units

Programs, package specifications and package bodies are all compilation units. One or more of them can appear in a source file. Package specifications must be compiled before the associated package bodies or any other units that import the package. When a package specification is compiled, the associated package body and any units that import the package are invalidated, and will need recompilation. This is particularly important to keep in mind when working with mutually dependent packages. Consider the packages in figure 8.

In this example package `a` imports `b`, and vice versa. The only way this can be compiled is if both package specifications are compiled before either package body. They could still appear in a single source file, but in this situation the most convenient arrangement is to have four source files, one for each compilation unit.

2.29 Built-In Procedures

SETL2 has a number of built-in procedures. Those without write parameters may be used as first class objects. They are all declared as part of a default scope, so the names may be hidden by any local declarations of variables with the same name.

```
package a;
   var va;
   procedure pa;
end a;

package body a;

   use b;

   procedure pa;
      print(vb);
   end pa;
end a;

package b;
   var vb;
   procedure pb;
end b;

package body b;

   use a;

   procedure pb;
      pa();
   end pb;
end b;
```

Figure 8: Package Dependencies

2.29.1 Atom Generation Procedure

The procedure `newat` generates a value not yet produced in the program. This value can be used as a unique key in maps, as pointers might be used in other languages.

`v := newat()` — Generates a unique atom value.

2.29.2 Arithmetic Functions

Most of the arithmetic functions accept numeric arguments and return a numeric result. None have write parameters so all may be used as first class objects.

`r := abs(v)` — `v` must be either an integer, a floating point number, or a one character string. If numeric, the return value will be the absolute value of `v`. If a character, the return value will be an integer corresponding to that character in the system's character set. At present, all the implementations use the ASCII character set so this function is portable. It should not be assumed that future implementations will use ASCII, however.

`r := sign(v)`	`v` must be either an integer or floating point value. The return value will be `1` if `v` is zero or positive and `-1` if it is negative.
`b := even(i)`	`i` must be an integer. The return value will be `true` if `i` is even and `false` otherwise.
`b := odd(f)`	`i` must be an integer. The return value will be `true` if `i` is odd and `false` otherwise.
`r := float(i)`	`i` must be an integer. The return value will be `i` as a floating point number. Note that because integers have infinite precision while floating point numbers are approximations the value of `r` may be different from `i`.
`r := sqrt(f)`	`f` must be a floating point number. The return value will be the square root of `f`.
`r := log(f)`	`f` must be a floating point number. The return value will be the natural logarithm of `f`.
`r := exp(f)`	`f` must be a floating point number. The return value will be e raised to the power `f`.
`r :=` *trigfunction*`(f)`	There are a variety of trigonometric functions available, each with the same form. `f` must be a floating point number, and the return value will also be floating point. The *trigfunction*'s available are: `cos sin tan acos asin atan tanh`
`r := atan2(f,g)`	`f` and `g` must both be floating point numbers. The return value will be the arc tangent of `f/g`.
`i := fix(f)`	`f` must be a floating point number. The return value will be the integer part of `f`. If the precision of `f` does not extend to the units position, then an error results and the program is terminated.
`i := floor(f)`	`f` must be a floating point number. The return value will be the largest integer less than or equal to `f`. If the precision of `f` does not extend to the units position, then an error results and the program is terminated.
`i := ceil(f)`	`f` must be a floating point number. The return value will be the smallest integer greater than or equal to `f`. If the precision of `f` does not extend to the units position, then an error results and the program is terminated.

2.29.3 Input - Output Procedures

SETL2 presently provides support for two distinct classes of files: text and binary. We think of a text file as a stream of lines made up of graphic characters. This stream may be a sequence of SETL2 values, or simply a sequence of character strings. If accessed as a sequence of SETL2 values, the input functions (`read`, `reada`, and `reads`) will work similarly to the SETL2 lexical analyzer. For

example, if the call `read(i,f,s1,s2,t)` is executed and the operator responds to the input prompt with the following string:

```
16#1f# 1.5e15 "hello world" {1,2,3,4,5} [1,2,3,4,5]
```

then the value of `i` will be the integer `31`, the value of `f` will be the floating point number `1500000000000000.0`, the value of `s1` will be the string `hello world`, the value of `s2` will be the set `{1,2,3,4,5}`, and the value of `t` will be the tuple `[1,2,3,4,5]`. Strings may be entered without quotes if they follow SETL2 rules for identifiers: they must begin with an alphabetic character, and consist only of letters, digits, and underscores. SETL2 operators and separators may not be read other than within quoted strings.

Binary files are also sequential, but contain an internal representation of SETL2 values. They can not be read with a simple text editor, but may be accessed more efficiently by SETL2 programs.

There are some restrictions on the I/O operations on internal SETL2 values, in particular atoms and procedures. These may be written to text files, but will not be written in a form in which they can be re-read. They will be human-readable, although cryptic. Those values may be written to binary files and re-read *within the same program execution.* If an attempt is made to read them in a different execution or within a different program, an error results and the program will stop. The lifetime of these values is restricted to the program execution which creates them.

`h := open(f,m)`

`f` must be a string whose value is a valid file name for the operating system being used and `m` must be a string whose value is one of the following:

`"text-in"`	File will be opened for input in text, or formatted, mode. It may then be accessed with `reada` or `geta`.
`"text-out"`	File will be opened for output in text, or formatted, mode. It may then be accessed with `printa`.
`"binary-in"`	File will be opened for input in binary mode. It may then be accessed with `getb`.
`"binary-out"`	File will be opened for output in binary mode. It may then be accessed with `putb`.

`open` prepares a file for access by other input/output procedures. It returns `h`, an atom, which must be used as a handle in subsequent procedure calls. If `h` has the value Ω, then the open failed, either because the file did not exist or the user did not have permission for the type of access requested.

`close(h)`

`h` must be a file handle created with a call to `open`. The file will be closed and the associated buffers will be returned to the operating system. The handle will no longer be accepted by input/output procedures.

`print(v1,v2...)`	Each of the values `v1,v2...` will be printed on the standard output device (usually the terminal). There will be no spaces or newlines between the values but a newline will be printed at the end.
`printa(h,v1,v2...)`	`h` must be a file handle created with a call to `open`. Each of the values `v1,v2...` will be written to that file. There will be no spaces or newlines between the values but a newline will be printed at the end.
`read(wr v1,wr v2...)`	Each of the arguments `v1,v2...` must be a valid left hand side. Values will be read from standard input and assigned to the variables in sequence. If an end of file is reached, all subsequent variables will have the value `om`, and a call to `eof()` will return `true`.
`reada(h,wr v1,wr v2...)`	`h` must be a file handle created with a call to `open`. Each of the arguments `v1,v2...` must be a valid left hand side. Values will be read from the file associated with `h` and assigned to the variables in sequence. If an end of file is reached, all subsequent variables will have the value Ω, and a call to `eof()` will return `true`.
`reads(rw s,wr v1,wr v2...)`	This is really a string scanning procedure, since it does no I/O, but it's behavior is closer to `read` than the other string scanning procedures. `s` must be a string, and each of the arguments `v1,v2...` must be a valid left hand side. Values will be scanned from the string `s` and assigned to the variables in sequence. If an end of string is reached, all subsequent variables will have the value Ω. The scanned portion of `s` will be removed. `reads` is most useful in converting numbers from character form to internal, but it works with sets, tuples, and identifiers as well.
`get(wr v1,wr v2...)`	Each of the arguments `v1,v2...` must be a valid left hand side. Lines will be read from standard input and assigned to the variables in sequence. If an end of file is reached, all subsequent variables will have the value Ω, and a call to `eof()` will return `true`.
`geta(h,wr v1,wr v2...)`	`h` must be a file handle created with a call to `open`. Each of the arguments `v1,v2...` must be a valid left hand side. Lines will be read from the file associated with `h` and assigned to the variables in sequence. If an end of file is reached, all subsequent variables will have the value Ω, and a call to `eof()` will return `true`.
`eof()`	`eof` returns `true` if the last read operation encountered and end of file and `false` otherwise.

2.29.4 String Handling Procedures

The procedures `str` and `char` produce character strings from other SETL2 values.

`s := str(v)` — `v` may be any SETL2 value. The return value will be the character string representation of `v`, as would be printed by `print`.

`s := char(i)` — `i` must be an integer. The return value will be the single character corresponding to that integer in the system's character set. At present, all the implementations use the ASCII character set so this function is portable. It should not be assumed that future implementations will use ASCII, however.

SETL2 provides a variety of string scanning procedures similar to those found in SNOBOL4.

`r := len(rw s,n)` — `s` must be a string and `n` must be an integer. `len` returns a string consisting of the first `n` characters of `s`. If the length of `s` is less than `n` the entire string is returned. The return string is removed from `s`.

`r := any(rw s,c)` — `s` and `c` must be strings. `any` returns the first character of `s` if that character is in `c`, and removes it from `s`. Otherwise it returns a null string and `s` is unchanged.

`r := notany(rw s,c)` — `s` and `c` must be strings. `notany` returns the first character of `s` if that character is *not* in `c`, and removes it from `s`. Otherwise it returns a null string and `s` is unchanged.

`r := span(rw s,c)` — `s` and `c` must be strings. `span` finds an initial substring of `s` consisting only of characters in `c` and returns it. The substring will be removed from `s`. `r` will be an empty string if the first character of `s` is not in `c`.

`r := break(rw s,c)` — `s` and `c` must be strings. `break` finds an initial substring of `s` consisting only of characters *not* in `c` and returns it. The substring will be removed from `s`. `r` will be an empty string if the first character of `s` is in `c`.

`r := match(rw s,c)` — `s` and `c` must be strings. `match` returns `c` if `c` is the initial substring of `s` and removes `c` from the front of `s`. Otherwise it returns a null string and `s` is unchanged.

`r := lpad(s,n)` — `s` must be a string and `n` must be an integer. `lpad` pads `s` with blanks until its length is `n`. If the length of `s` is greater than `n` then `s` itself is returned.

Each of the procedures above scan their arguments from left to right. There are corresponding procedures which perform similar functions but scan their arguments from right to left. Those procedures are:

`rlen rany rnotany rspan rbreak rmatch rpad`

2.29.5 Type Finding Procedures

In many programming languages the type of a variable can be determined at compile time and is enforced at run time. These languages are called *strongly* typed. SETL2 on the other hand is *weakly* typed: it's variables may be bound to values of more than one type during the execution of a program. Because of weak typing, it is frequently useful to test the type of a value at run time. The following procedures provide that capability.

`b := is_`*type*`(v)`

There are a variety of boolean functions which test whether a SETL2 value is of a certain type. They each return `true` if the value is of the given type and `false` otherwise. The functions available are:

```
is_atom    is_boolean    is_integer
is_real    is_string     is_set
is_map     is_tuple      is_procedure
```

`s := type(v)`

`v` may be any SETL2 value. The procedure will return a character string representation of the type of `v`. The possible return values are:

```
"ATOM"     "BOOLEAN"   "INTEGER"   "REAL"
"STRING"   "SET"       "TUPLE"     "PROCEDURE"
```

2.29.6 Environment Access Procedures

SETL2 provides a few procedures which return information about the program's environment.

`v := date()`

The return value will be a character string representation of the date maintained by the operating system.

`v := time()`

The return value will be a character string representation of the time maintained by the operating system.

`b := fexists(f)`

`f` must be a string whose value is a valid file name for the operating system being used. The return value will be `true` if the file exists and `false` otherwise.

`system(s)`

`s` must be a string whose value is a valid command for the operating system being used. The command will be passed to the operating system and execution suspended until the command is complete. **WARNING:** No error checking is done on the command string. This is not a safe procedure, so should be used sparingly!

`command_line`

`command_line` is not a procedure at all, but a built-in constant. The value of `command_line` will be a tuple of character strings containing the arguments passed to the interpreter on the command line.

3 SETL2 Operation Manual

3.1 Introduction

In this section we provide operating instructions for several implementations of SETL2. At present, implementations are available for the following machines and operating systems:

1. MS-DOS versions 3.0 and higher.
2. Apple Macintosh.
3. Sun 3 systems running Unix.
4. Sun 4 systems running SunOS.
5. DEC VAX systems running VMS.
6. DEC VAX systems running BSD Unix.

We have tried to provide a common user interface on all these systems, to the extent that is possible, so much of what follows is applicable regardless of the computer and operating system being used. Where there are computer or operating system dependencies we will give specific instructions for each.

Before describing the specific commands used to compile and execute SETL2 programs, we must describe libraries. While not demanded by the semantics of the language, libraries are the usual way of implementing packages, so it is important that the concept be clearly understood before proceeding.

3.2 Libraries

Most programming languages force the programmer to specify within the text of a program the names and types of all names imported from external modules to be linked with that program. When that is the case it is not necessary for the compiler to have access to those modules, only the linker needs such access. SETL2 on the other hand, is similar to Ada, which allows the programmer to import all the public names from another module simply by specifying the name of the module to import. In order to do that the SETL2 compiler must have access to all imported modules. There are at least two ways to implement this: The programmer could be forced to specify any linked modules on the compiler command line, or he could compile all modules into libraries, and let the compiler access them from the libraries. The implementations described here all use libraries, since that is much more convenient for the programmer.

A SETL2 library contains lists of public names, imported packages, object code, and some control information about compiled modules. There is no linker at present, programs are executed directly from the libraries and bound during program load.

The compiler uses libraries in two distinct ways. There is a single library which will be updated, and a list of libraries to be searched for any imported packages. The programmer can specify both the updated library and the input libraries on the compiler command line, in environment strings, or

can accept the system default. It is *very* unlikely that the system default will be acceptable, unless all SETL2 programs are kept in a single directory, and it is a nuisance to specify library names on the command line for each compilation. We highly recommend that as part of the installation process a command to set the appropriate environment strings is placed in your `.login` shell script (or `autoexec.bat` file, `login.com` file, or whatever is appropriate for the system being used).

3.3 Installation

There are three general steps to be performed before SETL2 may be used:

1. The executable files must be installed where commands are kept on the system being used.
2. A library must be created.
3. An environment string should be set containing the name of the library to be used.

For VMS and Unix users it is quite likely that the system administrator will have to install the executable files. Check with him or a local guru to see how to get started.

MS-DOS users must copy the executable files from the distribution diskettes to their hard disks. While it is possible to use SETL2 on diskette based systems, it is quite inconvenient, so we assume a hard disk with at least one megabyte of free space is available.

To install SETL2, all executable files must be loaded from the distribution diskettes to some directory in the search path for executable files. Set the current directory to something in your `PATH`, so that you will be able to execute these files from any directory. Then copy all the `.exe` files from the distribution diskettes to that directory. There should be three of those: `stll.exe`, `stlc.exe`, and `stlx.exe`.

One of the command files distributed with SETL2 is a library utility program. Eventually, this should perform a variety of functions, but at the moment it is only used to create empty libraries. To do that, change to any directory where you keep data files and enter the following command:

```
stll -c <library name>
```

A normal `<library name>` is `setl2.lib`.

Finally, you should set the environment string `SETL2_LIB` to tell SETL2 where your default library is. The command to do this is best placed in the startup command file for your system. For the operating systems supported now the commands are:

Operating System	Startup File	Command
MS-DOS	autoexec.bat	set SETL2_LIB `<library name>`
Unix	.login	setenv SETL2_LIB `<library name>`
VMS	login.com	define SETL2_LIB `<library name>`

3.4 Executing The Compiler

The command to compile a SETL2 source program is:

```
stlc <options> <file specifier> [<file specifier> ...]
```

You may use wild cards in specifying the files to be compiled. The following options are recognized:

`-s` Produce a listing of the program with line numbers and error messages. By default errors will be printed on standard output. If this option is given a listing file with the extension `.lis` will be produced.

`-t` *n* Set tab width to *n*. This is generally an unnecessary option. Error messages include both line and column numbers, and the column numbers will be incorrect if the source file uses tabs not set at eight column intervals. If an editor is used which can move the cursor to the line and column of an error, it will need that information. By setting this option any tab interval may be used. If line number information is sufficient (which is generally the case) this option can be ignored.

`-i` Disable implicit declarations. SETL implicitly declared any names for which it did not find an explicit declaration. SETL2 continued this by default, but for other than quick and dirty programs that may not be desireable. When this option is set all variables must be declared.

We do make implicit declarations by default, since we feel that the compile commands for larger programs will generally be in *make* files, which do not have to be changed very often. The compile command should be convenient for short programs, in which case implicit declarations will probably be desired. If you disagree with this philosophy, you can set the environment string to your own desired defaults.

`-f` Toggle use of intermediate files. By default, the compiler stores intermediate forms of the program on disk for MS-DOS systems and in core on Unix systems. This option reverses that setting. If you are using an operating system with virtual memory, you will probably find it faster to compile in core only. If you are using a PC, this is *not* recommended, since at present the compiler does not make use of extended or expanded memory and there is not much extra memory in a PC. The best thing you can do with expanded or extended memory is to use a RAM disk, and set the environment option to force intermediate files to be stored there.

`-l` *file name* Updated library name. This string must be a valid file name on the system you are using. If there are embedded spaces you must use quotes to make the file name a single string.

`-p` *file path* Library search path. This is a list of semicolon-delimited file specifiers which describe a list of files to be searched for any imported packages. Each specifier may contain wildcards if desired. You should note that the compiler will search only until it finds a package with the name given in a `use` clause. It will not check for duplicates. When wildcards are used, the order of search will depend on the disk subsystem (system calls are used to expand the file specifiers).

3.5 Executing The Interpreter

After the program is successfully compiled it may be executed with the command:

```
stlx <options> <program name> [<argument> ...]
```

CAUTION: When you execute a program, you give the name of the *program*, not the name of the file which contained it. For example: if the following program:

```
program this_is_a_junk_program;

   var some_trash;

       ...

end this_is_a_junk_program;
```

is in the file `junk.stl`, then you would compile that program with the command

```
stlc junk
```

but you would execute the program with the command

```
stlx this_is_a_junk_program
```

The arguments after the program name will be gathered into a tuple, and will be available to the SETL2 program as `command_line`.

The options available are:

`-a` *mode*	This sets the assert mode. There are two possible values for *mode*: `f` is used if failing assertions should stop the program but succeeding assertions should be skipped, and `l` is used if succeeding assertions should be logged.
`-l` *file name*	Updated library name. The library is not actually updated, this is provided for consistency with the compiler. This library will be searched before those in the search path.
`-p` *file path*	Library search path. This is a list of semicolon delimited file specifiers which describe a list of files to be searched for any imported packages. Each specifier may contain wildcards if desired. You should note that the compiler will search only until it finds a package with the name given in a `use` clause. It will not check for duplicates. When wildcards are used, the order of search will depend on the disk subsystem (system calls are used to expand the file specifiers).

3.6 Environment strings

The following is a list of useful environment strings recognized by the SETL2 compiler and interpreter.

`SETL2_LIB`	This lets you place the name of your update library in the environment, rather than specifying it on the command line. You should probably place a command in your `.login` script or `autoexec.bat` file to set this for you when you log on.
`SETL2_LIBPATH`	This is a list of libraries which will be searched for imported packages. The default is empty, which will probably be OK generally. It is just a string of comma-delimited file specifiers, which may include wildcards.
`SETL2_TMP`	This string is a prefix used to specify where temporary files should be stored. It is primarily useful on a PC, where you should use this to hold the letter of your RAM disk, if you have one. For example, if your RAM disk is logical drive `d`, you would place in your `autoexec.bat` file the command `"set SETL2_TMP d:"`.
`STLC_OPTIONS`	This is a string of options which will be read in front of the command line by the compiler. You can use this to change the default for implicit declarations, tab width, or anything else.
`STLX_OPTIONS`	This is a used just like the previous string, but is used by the interpreter.

3.7 Acknowledgements

Many of the changes made in SETL2 were motivated by comments and ideas voiced in the SETL meetings at New York University. I would like to thank all who attended those meetings and participated in the discussions, in particular Robert Dewar, Fritz Henglein, Bob Paige, Ed Schonberg, and Matthew Smosna.

I would also like to thank Jack Schwartz, who offered many helpful comments and suggestions, and gave the system its most thorough test.

A A Random Number Generator

SETL provided a built-in operator to produce random numbers, but SETL2 does not provide such an operator. An alternative in SETL2 is to create a package to do the same thing, which is what we will do here. It is not sufficiently powerful for all users, but serious users of random number generators can modify this fairly easily. See [Knu81] for discussion of pseudo-random numbers.

There are two things to make note of in this example. The first is the general idea of using packages to encapsulate a function, and the second is the way in which the data for a specific random number stream is hidden within the package. We used atoms to identify streams, and these atoms are the only values accessible from outside the package.

```
--
--  RANDOM NUMBERS
--  ==============
--
--  This package is meant to replace the "random" built-in procedure in
--  SETL.  It is somewhat different conceptually from that procedure.
--
--  We allow the creation and access of 'streams' of random numbers.  To
--  create a stream, call start_random passing it some kind of source and
--  an initial seed.  The source should be one of the following:
--
--     1.  An integer.  In this case we return integers from 1 to that
--         integer.
--
--     2.  A real.  We return reals from 0.0 to that real.
--
--     3.  A set.  We return random elements from that set.
--
--     4.  A tuple.  We return random elements from that tuple.
--
--  The seed may be an integer, or om.  If it is om we use the time as
--  the initial seed.
--

package Random_Numbers;

   procedure Start_Random(Source,Seed);

   procedure Random(Handle);

end Random_Numbers;

package body Random_Numbers;

   const  Modulus      := 2 ** 64 - 59,
          Multiplier   := 2 ** 60 - 93,
          Increment    := 2 ** 15 - 19;

   var    Stream_Set   := {},
          Source_Map   := {},
          Seed_Map     := {};
```

```
--
--  Start_Random
--  ------------
--
--  This procedure is called to initialize a stream of random numbers.
--  It returns a handle which is used to access the stream.
--

procedure Start_Random(Source,Seed);

   var Handle;

   --
   --  Allocate a handle for this stream.
   --

   Handle := newat();
   Stream_Set with := Handle;

   --
   --  Set the initial seed.  If we get one from the caller, use that.
   --  Otherwise use the time.
   --

   if Seed = om then

      t := Time();
      hour := t(1 .. 2);
      reads(hour,num_hour);
      minute := t(4 .. 5);
      reads(minute,num_minute);
      second := t(7 .. );
      reads(second,num_second);
      Seed_Map(Handle) := num_hour * 60 ** 2 +
                          num_minute * 60 +
                          num_second;

   elseif not is_integer(Seed) then

      print("Invalid seed in Start_Random => ",seed);
      stop;

   else

      Seed_Map(Handle) := Seed;

   end if;

   --
   --  Save the source in a map.
   --

   case type(Source)

      when "INTEGER", "REAL", "TUPLE" =>
```

```
            Source_Map(Handle) := Source;

         when "SET" =>

            Source_Map(Handle) := [x : x in Source];

         otherwise =>

            print("Invalid source in Start_Random => ", Source);
            stop;

      end case;

      return Handle;

end Start_Random;

--
--  Random
--  ------
--
--  This procedure returns a single random number (or element from set
--  or tuple).
--

procedure Random(Handle);

      --
      --  Validate the handle.
      --

      if Handle notin Stream_Set then

         print("Invalid handle for Random");
         stop;

      end if;

      --
      --  Find a random integer (linear congruential method).
      --

      New_Seed := (Seed_Map(Handle) * Multiplier + Increment) mod Modulus;
      Seed_Map(Handle) := New_Seed;
      Source := Source_Map(Handle);

      --
      --  Return the random number.
      --

      return case type(Source)
         when "INTEGER" =>
            (New_Seed mod Source_Map(Handle)) + 1
         when "REAL" =>
            float(New_Seed) / float(Modulus) * Source_Map(Handle)
         when "TUPLE" =>
```

```
            Source((New_Seed mod #Source) + 1)
         end case;

      end Random;

   end Random_Numbers;
```

B The Stable Assignment Problem

The following program is an implementation of the Gale-Shapely stable assignment problem (see [SDDS86] for the SETL version of this program). It is a good illustration of the power of set-forming expresssions.

```
--
--  STABLE ASSIGNMENT PROGRAM
--  =========================
--
--  This program matches students with colleges in such a way that the
--  following three conditions are satisfied:
--
--     1.  No college accepts more than quota(c).
--     2.  A college never admits a student if it has filled its quota
--         and there exists an unassigned student to whom the college is
--         acceptable and the college prefers.
--     3.  There is not situation in which two students each prefer the
--         other's college, and each college prefers the other's student.
--
--  The algorithm is due to David Gale and Lloyd Shapley.
--

program gale_shapley;

   -- colleges

   const A := "NYU", B := "Harvard", C := "Princeton", D := "MIT";

   -- student preferences

   stud_pref := {[1,[A,B,C]],[2,[B,C,A,D]],[3,[C,A,B]],[4,[B,A,C]]};

   -- college preferences

   coll_pref := {[A,[1,2,3,4]],[B,[4,3,2,1]],[C,[2,4,3]],[D,[1,2,3]]};

   -- college quotas

   quota := {[A,2],[B,1],[C,1],[D,2]};

   -- perform the assignment and print results

   print(assign(stud_pref,coll_pref,quota));

   --
   --  Assign
   --  ------
   --
   --  Make the stable assignment.
   --
```

```
procedure assign(rw stud_pref,coll_pref,quota);

   colleges := domain quota;
   active := {[c,[]] : c in colleges};  -- active list by college
   applicants := domain stud_pref;      -- initialize applicant list

   -- we may need as many rounds as there are colleges

   for j in [1 .. #quota] loop

      new_applicants := applicants;

      -- each student who has a college to apply to does so

      for s in applicants | stud_pref(s) /= [] loop

         first_choice fromb stud_pref(s);
         active(first_choice) with:= s;
         new_applicants less:= s;

      end loop;

      applicants := new_applicants;

      -- drop all over quota applicants

      for c in colleges | #active(c) > quota(c) loop

         active(c) := pref_sort(active(c),coll_pref(c));

         for k in [quota(c)+1 .. #active(c)] loop
            applicants with:= active(c)(k);
         end loop;

         active(c) := active(c)(1 .. #active(c) min quota(c));

      end loop;

      if not (exists s in applicants | stud_pref(c) /= []) then
         exit;
      end if;

   end loop;

   return [active,applicants];

end assign;

--
--  Pref_sort
--  ---------
--
--  Sort applicants by college choice.
--

procedure pref_sort(apvect,order);
```

```
      applicants := {x : x in apvect};
      return [x in order | x in applicants];

   end pref_sort;

end gale_shapley;
```

C A Five Function Calculator

The following program is a simple five-function calculator, which accepts expressions from the standard input and evaluates them. It illustrates a possible use of procedure values. In this program, procedures are embedded in tuples and maps.

```
--
--  CALCULATOR
--  ==========
--
--  This is a simple five function calculator.  It handles valid SETL2
--  expressions made up of +, -, *, /, and **.  The error handling is
--  rather crude.
--

program Calculator;

   var Operator_Info;

   --
   -- Operator information map.  The general form is
   --
   --     [operator, [in-stack-priority,in-coming-priority,handler]]
   --

   Operator_Info := {["(",[0,4,om]],
                     [")",[om,om,om]],
                     ["+",[1,1,Binop_Plus]],
                     ["-",[1,1,Binop_Minus]],
                     ["*",[2,2,Binop_Times]],
                     ["/",[2,2,Binop_Divide]],
                     ["**",[3,4,Binop_Power]]};

   --
   --  main loop -- get a line, find the result, print it
   --

   while true loop

      get(Input_Line);

      if eof() then
         exit;
      end if;

      print(Input_Line," = ",Solve(Input_Line));

   end loop;

   --
   --  Solve
   --  -----
   --
   --  This procedure accepts a character string containing an
   --  expression, evaluates it, and returns the result.
```

```
--

procedure Solve(Input_Line);

   Work_Line := Input_Line;
   Operand_Stack := [];
   Operator_Stack := [];

   while #Work_Line > 0 loop

      -- skip white space

      span(Work_Line,""+/[char(i) : i in [0 .. abs(" ")]]);

      -- pick off the next operand

      if Work_Line(1) in {str(i) : i in [0 .. 9]}+{"{","[","\""} then

         reads(Work_Line,Token);
         Operand_Stack with:= Token;
         continue;

      end if;

      -- we didn't find an operand, we had better find an operator

      Token := "";

      while #Work_Line > 0 and
            Operator_Info(Token+Work_Line(1)) /= om loop
         Token +:= throwaway fromb Work_Line;
      end loop;

      if Operator_Info(Token) = om then
         print("Invalid operator => ",Token);
         stop;
      end if;

      --
      --  when we find a closing parenthesis, we evaluate until we
      --  find the corresponding opening parentheses
      --

      if Token = ")" then

         while #Operator_Stack > 0 and
               Operator_Stack(#Operator_Stack) /= "(" loop

            Operator frome Operator_Stack;

            if #Operand_Stack < 2 then
               print("Invalid expression");
               stop;
            end if;

            right frome Operand_Stack;
```

```
            left frome Operand_Stack;

            Operand_Stack with :=
               Operator_Info(Operator)(3)(left,right);

         end loop;

         if #Operator_Stack = 0 then
            print("Invalid expression");
            stop;
         end if;

         Operator frome Operator_Stack;
         continue;

      end if;

      --
      --  we evaluate while the stack priority of the top operator is
      --  greater than the incoming operator
      --

      while #Operator_Stack > 0 loop

         Operator frome Operator_Stack;

         if Operator_Info(Operator)(1) >= Operator_Info(Token)(2) then

            if #Operand_Stack < 2 then
               print("Invalid expression");
               stop;
            end if;

            right frome Operand_Stack;
            left frome Operand_Stack;

            Operand_Stack with :=
               Operator_Info(Operator)(3)(left,right);

         else

            Operator_Stack with:= Operator;
            exit;

         end if;

      end loop;

      Operator_Stack with:= Token;

   end loop;

   --
   --  we've exhausted the input string, evaluate any remaining
   --  operators
   --
```

```
      while #Operator_Stack > 0 loop

         Operator frome Operator_Stack;

         if Operator = "(" then
            print("Invalid expression");
            stop;
         end if;

         if #Operand_Stack < 2 then
            print("Invalid expression");
            stop;
         end if;

         right frome Operand_Stack;
         left frome Operand_Stack;

         Operand_Stack with :=
            Operator_Info(Operator)(3)(left,right);

      end loop;

      if #Operand_Stack /= 1 then
         print("Invalid expression");
         stop;
      end if;

      --
      --  the result is on the top of the operand stack
      --

      return Operand_Stack(1);

   end Solve;

   --
   --  Operator Procedures
   --  -------------------
   --
   --  The following simple procedures just implement the primitive
   --  functions of the calculator.
   --

   procedure Binop_Plus(left,right);
      return left + right;
   end Binop_Plus;

   procedure Binop_Minus(left,right);
      return left - right;
   end Binop_Minus;

   procedure Binop_Times(left,right);
      return left * right;
   end Binop_Times;
```

```
   procedure Binop_Divide(left,right);
      return left / right;
   end Binop_Divide;

   procedure Binop_Power(left,right);
      return left ** right;
   end Binop_Power;

end Calculator;
```

References

[Ada83] United States Department of Defense, New York, NY. *Ada Programming Language Reference Manual*, 1983. (ANSI/MIL-STD-1815A).

[Bar82] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley Publishing Company, London, 1982.

[Knu81] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1981.

[Mac87] B. J. MacLennan. *Principles of Programming Languages.* Holt, Rinehart, and Winston, New York, NY, 1987.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, NY, 1986.