Index

declarations
procedure declarations
end pckgname;
package body pckgname;
use section
hidden declarations
procedure definitions
end pckgname;

The package is split into two parts: the package specification and the package body. Within the specification part, the *declarations* may contain **var** and **const** declarations. These variables will be accessible in any program, or subprocedure, that imports the package. The *procedure declarations* contain the **procedure** lines of the definitions that are given in the body of the package, and these procedures will then be accessible to any program that imports the package. Note that the package specification, which is the *visible* portion of the package, is terminated with an **end** statement. Within the body of the package, a use section may be invoked to import packages in a nested fashion; *hidden declarations* contain **var** and **const** declarations for variables that will be visible within the package body but not in programs that import the package, and *procedure definitions* contain the actual procedures, as described before. Other procedures may also be defined within the package body, in addition to those declared in the *procedure definitions* part, but these procedures will be visible only within the package.

Package specifications and package bodies may be compiled separately, and packages can thereby recursively call one another.

17. Using the interpreter

SETL2 may be obtained by anonymous ftp to cs.nyu.edu under the directory pub/setl2. There are interpreters for SUNs, VAXes, PC's running MS-DOS, Macintoshes, and other platforms.

When using SETL2, you must first create a library using the command:

```
stll –c setl2.lib
```

Other names may be given for libraries, but this name is the most common. SETL2 modules (programs or packages) may then be compiled using:

```
stlc file
```

where the file gives the file name containing the program or package. The compiled units will be stored in the library file setl2.lib under the working directory. Other libraries may be used, but have to be named by command-line options.

To execute a program that has been successfully compiled, the command

```
stlx prog_name
```

is given, where prog_name is the name of the program from the **program** statement (and not the name of the file containing the source file for the program). The compiled program in the library setl2.lib is then interpretted and executed. Again, other libraries may be used; see the manual pages or instructions for the stll, stlc, and stlx commands.

Within the body of the procedure, the **return** statement is used to return control to the caller, and provide a returned value. The format is:

```
return expression;
return;  -- means return om;
```

If no **return** statement is executed, the procedure returns after executing the final statement in the procedure block, the returned result being **om.**

Procedures may be called either as a statement (in which case the returned value is ignored), or as a function in an expression, in which case the returned value is the value of the function call:

```
prname(10,120,30);        -- call as a statement
a := b + prname(1,2,3);   -- call as a function
```

Multiple values can conveniently be returned from a procedure by using tuple formers and tuple assignment:

```
[x,y,z] := prname(2,3);
...
procedure prname(arg1,arg2);
...
return [10,20,a+b];
...
end prname;
```

Procedures in SETL2 may be called recursively (i.e. they may call themselves directly or indirectly). All identifiers which are local to the procedure are saved recursively to avoid confusion between values at different recursion levels:

```
procedure factorial(arg);
    if arg=1 then
        return 1;
    else
        return arg * factorial(arg−1);
    end if;
end factorial;
```

Procedures declared in the manner given above allow only call by value. The following extended form of procedure definition allows one or more arguments to be specified as value receiving:

```
procedure name (type arg, type arg, type arg, ...);
```

where *type* is one of the following:

rd The argument is read only. This is the default value obtained if *type* is omitted, and causes the argument to be passed by value in the manner already described. Within the body of the procedure the parameter name is treated as a variable, but modifications to the value of such variables do not affect the arguments in the call.

rw The argument is read/write. The value of the argument will be passed as the initial value of the parameter. The parameter identifier is treated as a variable in the body of the procedure, and may be reassigned a new value. On return from the procedure, whatever final value is in this variable at the point when the **return** statement is executed is transmitted back as the new value of the calling argument (which must have the proper syntax for an assignment left hand side).

wr The argument is write only. On entry to the function, the initial value of the corresponding parameter is set to **om**. The parameter is treated as a variable and assigned to the calling argument on return in the same manner as described for a read/write parameter.

We recall that a procedure name may be used as the value of a procedure variable providing all of the parameters to the procedure are of type **rd**.

16. Packages

As noted before, a **use** statement may import a package of declarations and procedure definitions into a program unit. A package is compiled separately, and so is contained in a separate file (or files). The form of a package, which is similar to a program unit, is:

```
package pckgname;
```

const *name:=value,name:=value, ...* ;
var *name,name,...*;
sel *name*(1), *name*(2), ... ;

The **const** statement defines identifiers to be associated with specified constant values, and initializes those values. Such constant identifiers cannot appear on the left side of assignments. The right hand sides in **const** declarations may be denotations, previously declared constant names, or sets or tuples of constant values. The **var** statement merely names a list of variables which are assigned an initial **om** value as usual, and will be visible in all nested procedures. Optionally, the **var** declarations can also give initial values for the variables, as in:

var name1:=falue1,name2:=value2,...;

The initialization of the variables simply replaces executable assignments that would otherwise come before all other executable statements.

All variables and constants declared in the global declarations section may be accessed by any part of the program, including any nested procedures, until hidden by a declaration of an object with the same name. Variables that are hidden by nested declarations can still be accessed using a ‘‘dot-construct,’’ as in:

```
procedure a( );
        var x1;
        body
        procedure b( );  -- Nested subprocedure
                var x1;  -- Hides x1 defined in procedure a
                body...;
                t1 := a.x1;          -- Accesses x1 from procedure a
                t2 := x1  -- Accesses x1 from procedure b
                rest...;
        end b;
end a;
```

Note that the declaration of x1 in b hides the x1 variable that has been declared in a, but that the x1 of a is still accessible by the notation ‘a.x1’.

Any identifiers which are referenced in the program block, but which are not included in the global declarations are private to the program block and may not be accessed by procedure definitions at deeper levels, unless passed via parameters. A similar statement is true for identifiers at any level of nesting within a procedure.

Finally, selector declarations, using **sel**, come after the other declarations, but before the executable statements of the program body, and denote names for elements of aggregate types, as described in Section 8. Selectors are accessed using a ‘‘dot-notation,’’ and are active at all nested levels below the level of declaration. Note that the dot-notation for selectors uses an identifier declared as a selector after the dot, whereas the dot notation for access of hidden objects uses an identifier declared as a variable or constant after the dot.

15.3. Procedure Definitions

The form of a procedure definition, as noted above, is basically the same as the form of a program, but without the **use** section. Procedure definitions must follow the executable statements in the unit, (i.e., come after the body of the program or parent procedure), and are bracketed by **procedure** and **end** statements, as in:

```
procedure prname(arg1,arg2,...);
...
end prname;
```

The list of parameters arg1, arg2,... are identifiers which are assigned the argument values when the procedure is called. These names are strictly local to the procedure, and must be different from the names of any declared identifiers in the parent program or procedure. Within the procedure definition (i.e., prname in the example), these names act as ordinary identifiers that have been declared as **var**’s with values initialized by the call to the procedure. It is permissible to reassign new values to these identifiers in the body of the procedure, but such assignments do not affect the parameters in the call since the call is a call by value. If there are no arguments, then the parentheses surrounding the argument list may be omitted, or a null list may be retained.

The declaration section, if it is present, contains **var**, **const**, and statements in the same format as that used in the program declaration section. The initialization of local variables to **om**, whether declared by **var** (without initialization) or implicitly declared, occurs on each entry to the procedure.

```
when 2,4,7 => print(i+1);
when 0,6,9 => null;          -- do nothing in these cases
otherwise => print("no good");
end case;
```

15. Program Form

SETL2 is a block-structured language. Thus the main program can contain procedures, which in turn can contain procedures, which in turn can contain nested procedures, etc. Variables and objects at one level are local to the unit in which they appear, and are visible in nested lower levels only if:

(1) The variables have been passed as procedure parameters; or

(2) The variables or objects are declared as **var**'s or **const**'s or in procedure definitions, and have not been hidden by subsequent declarations at lower levels.

Accordingly, each level contains the program or procedure name, the declarations, the main body, and the subprocedures. Within any program or procedure, the code may make use of objects declared within that unit including procedures, objects that are implicitly declared (not all variables must be declared), objects that are declared in higher levels including procedures that are defined in higher levels, (providing the names of the objects in higher levels are not hidden by objects with the same name in intervening levels), procedures imported from **use** packages, and builtin procedures.

A complete program in SETL2 has the form:

```
program name;
(use section)
(constant, variable and selector declarations)
...
(main program block)
...
(procedure definitions)
...
end name;
```

The form of a procedure, such as a procedure in the *procedure definitions* section, is similar:

```
procedure name(param1,param2,...);
...
(constant, variable and selector declarations)
...
(main procedure block)
...
(subprocedure definitions)
...
end name;
```

15.1. Use section

A package is a collection of variables, constants, selectors, and procedures which may be compiled separately and imported into programs and other packages, and has a similar structure to a program unit, as discussed in Section 16 below. A program unit may make use of objects in a package, just as though the objects (variables, constants, selectors, and procedures) were defined within the program, by means of the **use** declaration:

```
use package_name;
```

A **use** statement may contain multiple package names, separated by commas:

```
use package_name1, package_2, package_3;
```

15.2. Constant, variable, and selector declarations

Constants and variables may be declared in order to be visible in procedures nested in levels below. Together with selector declarations, the declarations take the form:

Real values are converted either in fixed point format or exponent notation, depending on the range. The number of digits is chosen to be appropriate to the accuracy with which real values are stored.

String

The handling of strings depends on whether they appear directly in the print list, or as elements of tuples or maps. If they appear in the print list, they are printed literally, without surrounding quotes, and without any special treatment of internal quotes. Strings which appear as elements of tuples or sets are treated in a different manner. If the string values are of the form of identifiers (starting with a letter and containing only letters, digits and the underline), then the value is printed literally. All other string values appearing as elements of sets or tuples are printed with surrounding quotes, and any internal quotes are printed as two successive quotes.

Boolean

The boolean values **true** and **false** are printed as TRUE and FALSE respectively.

Om (undefined)

An undefined value is printed as ‘<om>’.

Tuples

Tuple values are output as a series of values separated by single blanks and surrounded by tuple brackets [ ], or, in implementations which do not have these characters, simple parentheses. The values within the tuple are converted individually, strings being output with surrounding quotes unless they have the form of identifiers.

Sets

Sets are output in the same manner as tuples except that the list of values is surrounded by set brackets { }.

From this description, it can be seen that the format used is essentially exactly compatible with the input format accepted by **read** with the exception that strings appearing directly in the **print** list are printed without quotes. This discrepancy allows the convenient output of titling information as in:

print("Value of", a, " +", b, " =", a+b);

Each new call to **print** causes a new line to be started. Blanks are inserted to separate consecutive values, and if the value to be printed does not fit one a single line, line returns are inserted in an attempt to make the output as readable as possible. Calling **print** with a null argument list generates a blank line:

print( ); **- -** Print blank line

12. Assert statement

The assert statement has the form:

assert *test*;

where *test* is any expression that evaluates to a boolean value. Depending on a command-line option at run-time, the assert statement may cause the program to stop if the test does not evaluate to **true**.

13. Stop Statement

The **stop** statement:

stop;

may be executed at any point in the program and causes immediate termination of execution. Execution termination can also result simply from executing the last statement of the main program block.

14. Null Statement

The **null** statement:

null;

has no effect and thus acts as a no-operation statement. It is sometimes useful in connection with conditional statements in the case where no actions are to be performed under some conditions, for example in:

case i
when 1,3,5 => **print**(i);

reread from the file.

11.3. Command line

Data can be obtained from the command line by using the reserved word **command_line**, which is a tuple all of whose elements are strings obtained from parameters that follow the program name on the SETL2 execution line. Thus **command_line**(1) is a string giving the first parameter, **command_line**(2) is the second string, etc. Command line parameters frequently make use of the **unstr** function:

v1 := **unstr**(**command_line**(1));

Then when the execution line 'stlx program 1.0' is given (see Section 17), the value of v1 will be set to the real value 1.0.

11.4. Format

The data that is read or written by I/O procedures other than binary I/O procedures will be formatted, using simple and natural formatting conventions. For example, the **read** procedure is used to read values from data lines from the standard input source in stream mode:

read(lhs1,lhs2,...);

This procedure call reads successive values separated by blanks, commas, tabs, or newlines. Data values are entered in the following form:

- Integers are entered as strings of digits preceded by an optional sign.
- Reals are entered in the same format as real denotations except that an optional preceding sign is permitted.
- Strings can be entered in the same format as string constants in SETL, including the surrounding quotes. Newlines must be entered in the form '\n', and double quotes may be entered as ' In addition, strings which have the form of SETL identifiers (i.e. start with a letter and contain only letters, digits and the underline character, and are not reserved words) may be entered without surrounding quote marks.
- ''Om'' or ''om'' in the input stream causes the corresponding value to be set to undefined **(om).**
- The values **true** and **fals**e are entered as TRUE and FALSE (or true and false, either upper or lower case may be used).
- Set values are input using { } brackets. The list of values in the set are in the format described in this section, and may be separated either by blanks or by commas.
- Tuple values are input in a similar manner using [ ] brackets.

The arguments used in the call to **read** are any valid assignment left hand sides, and the effect of the **read** call is to input the appropriate number of items from successive input lines (as required) and make the assignments in sequence. If an end-of-file is read, the remaining variables are set to **om**. The boolean function **eof**( ) may be used as a test following a **read** call. This test succeeds if the last read attempted to read past the end of file, and fails if the end of file was not encountered.

The **get** function may be used to read input lines one line at a time as strings. This allows string data to be entered without surrounding quote marks, and the program determines the required format of the input. Thus

get(s1,s2);

will cause the next two input lines to be read from the standard input file and assigned as string values respectively to s1 and s2. **Eof** is set as described above to indicate whether or not the call attempted to read past the end of file, although the strings for the lines not read will be empty strings, rather than being set to **om**.

Printed output is generated with the **print** function, which gives a list of items to be printed in stream mode:

print(expression,expression, ... expression);

Each expression is evaluated and printed in a manner appropriate to its datatype as follows:

Integer

The integer value is converted to a string of decimal digits of appropriate length with no leading zeroes (except in the case of zero itself). A preceding minus sign is used if the value is negative (but positive values do not generate a plus sign).

Real

11.1. I/O Procedures

There are three distinct types of input/output in SETL2: text, binary, and string. In addition, text I/O can utilize procedures which assume that I/O takes place using the standard input and standard output.

Text Input/Output

Text input/output is intended for direct input of human prepared input, and output of human readable printout. This type of input/output is oriented to transmission of individual values in string form. Most values will be written out in a form that they can be read back in; however, atoms and procedures can not be read back in, and the string representations for atoms and procedures will only identify the type of the variable and a unique number associated with the variable. Further, the associated number need not be consistent from one execution of the program to the next. The available procedures for text I/O are:

print(v1,v2, ...);	-- For printing values to the standard output
read(v1,v2, ...);	-- For reading values from the standard input
get(v1,v2, ...);	-- For reading lines from the standard input as strings
printa(handle,v1,v2, ...);	-- For printing values to a file -
	-- handle must have been openned for text-out.
reada(handle,v1,v2, ...);	-- For reading values from a file -
	-- handle must have been openned for text-in
geta(handle,v1,v2, ...);	-- For reading lines from a file as strings -
	-- handle must have been openned for text-in
eof( )	-- Returns true if an eof has just been read.

Binary Input/Output

Binary input/output deals with SETL2 values of any type and transmits then in a special efficient internal form. Binary output is only used to output values which are to be read into either the same SETL2 program, or some other SETL2 program, using binary input statements. However, atoms and procedure types that are written out to a binary file will only have meaning within that execution of the program — another program or another execution of the same program might assign a different representation to these variables. There are only two procedures:

putb(handle,v1,v2, ...);	-- For writing values in binary form to a file
getb(handle,v1,v2, ...);	-- For reading values in binary form from a file

In both cases, the variable handle must be associated with a file, and be openned for binary I/O (see ‘‘Openning files’’ below).

String Input/Output

String input/output reads and writes values to internally represented strings. An input operation obtains a record as a string, and an output operation outputs a string as a single record. Record input/output is especially useful for communication with programs written in languages other than SETL2, and is also used to read directly prepared input if such input is more naturally treated as strings, rather than as individual items as in the text case. That is, text I/O might be used to read lines of text, converting them into strings, and then string operations and string I/O can be used to extract values. There are about two and a half procedures:

reads(str, v1,v2, ...)	-- Reads values from string str
unstr(str)	-- Returns value of string str
str := **str**(v);	-- Converts a single value into a string

Reads reads multiple values from a string, whereas **unstr** assumes that the string is a single object. The procedure **str** produces the string that would be printed by **print** if it were used to print a single value to the standard output. There is also a **char** function to convert an integer into a single-character string, using the ascii conversion code.

11.2. Openning files

File I/O, as described above, requires that a file be openned. There are two procedures, **open** and **close**. The are described by:

handle := **open**(filename,mode);	-- Opens file filename
close(handle);	-- Closes file associated with handle

In both cases, the handle is a variable of type atom that becomes associated with the file as a result of the **open** procedure. The variable mode is a string from among the following choices: "text-in", "text-out", "binary-in", and "binary-out". Note that a file openned for output must be closed and reopenned for input before values can be

In the case of the **forall** test, the loop is terminated as soon as the test fails, leaving the iteration variables set to the values which caused the failure. If the test succeeds for all loops, then the iteration variables are set to **om** and the **forall** succeeds.

The following are some examples of the use of these quantified tests:

```
s := {1,2,10,20};
t := [1,2,10,20];

if exists x in s | x > 3
  then ...                 -- will be executed with x = 20 or 10
  else ...                 -- will not be executed
end;

if exists x in t | x > 3
  then ...                 -- will be executed with x = 10
  else ...                 -- will not be executed
end;

if forall x in a | x < 30
  then ...                 -- will be executed with x = om
  else ...                 -- will not be executed
end;

if exists x in t | x > 30
  then ...                 -- will not be executed
  else ...                 -- will be executed with x = om
end;

if forall x in t | x < 10
  then ...                 -- will not be executed
  else ...                 -- will be executed with x = 10
end;
```

10.5. Compound Operators

Another specialized form of loop in SETL2 is the compound operator. A compound operator can be formed from any binary operator by appending a slash / to the name of the operator. Such a compound operator can be used in one of two expression forms:

bop/ exprs
expre bop/ exprs

The effect is to apply the binary operator *bop* to the sequence of element values *e1,e2...* in *exprs*, which must be a set or tuple, as follows:

bop/ exprs	means	*e1 bop e2 bop e3 ...*
expre bop/ exprs	means	*expre bop e1 bop e2 ...*

If the compound operator form is used with a null set or null tuple, then the result is expre in the second form and **om** in the first form where expre is omitted. Thus expre typically functions as an identity element in the second form. The following examples show how the compound operator forms can be used:

```
+/t        -- sum of values in tuple
0+/t       -- same, but 0 rather than om for [ ]
*/ [a in s | 3 in a]
" "+/t     -- builds string from tuple of characters
```

11. Input/Output

statement1; ... ;
exit *expression*;
statements; ... ;
end loop;

The **exit** statement will most likely contained in a conditional branch in the loop, since it is not likely to be executed on every iteration. The value of x will become the value of *expression* upon execution of the exit statement. As a shorthand, alternate forms of the **exit** statement are:

exit when *test*;
exit *expression* **when** *test*;

which are respectively the same as:

if *test* **then exit**;
if *test* **then exit** *expression*;

If an **exit** statement is never executed within a loop, then the value of the loop expression is **om**.

A loop expression may also be used with **while** loops and **until** loops: wherever an **exit** statement is valid. Note especially that the value of the bound variable (s in the above example) is undefined upon exiting from the loop. Only the expression evaluated in the exit statement is carried from the loop.

10.3. Set & Tuple Formers

So far, we have formed sets by enumerating the elements. The set former is a special form of a loop which computes a set value with an iteration. The form is:

{*expression* : *iterator*}

The iterator has exactly the same form as a loop iterator, and can include a conditional test. The effect is to iterate the calculation of the expression, and build a set from the values. The most frequently used form involves a set or numeric iterator as shown by the following examples:

{n : n **in** {1..100}} -- integers from 1 to 100
{[x**2,x] : x **in** {1..5} -- square root map
{a : a **in** y | a>5} -- elements > 5

The following abbreviation is permitted, allowing expression to be elided where it corresponds exactly to the loop variable of an iterator provided that the iterator contains a '|' (conditional) clause:

{a **in** expr | c} -- {a : a **in** expr | c}
{a **in** [x,y..z] | c} -- {a : a **in** [x,y..z] | c}

If two or more iterations produce the same value in a set former, then only one copy of the value is placed in the set. If the iterator terminates after zero iterations, i.e. expression is never calculated, then the result is a null set. An attempt to place **om** into a set is ignored.

Tuple formers are identical in form, but written with tuple brackets, rather than set brackets. For a tuple former, the successive values, which may include duplicated values, are placed into the tuple in sequence:

[0 : i **in** [1..100]] -- tuple of length 100, all 0
[x **in** s | x < 0] -- tuple of negative elements of s

10.4. Quantified Tests

Two special forms, called quantified tests, can be used in contexts requiring a test, or to generate one of the two values TRUE or FALSE in other contexts:

exists *iterator* | *test*
forall *iterator* | *test*

The **exists** test executes an implied loop specified by the *iterator*, which has the same form as a loop iterator without a conditional test (which instead is required as the quanifier test). On each loop, the test after the '|' is evaluated. If the test succeeds, then the loop is immediately terminated, leaving the loop variable set to the value which caused the test to succeed, and the **exists** test itself succeeds. If the specified test fails for all iterations of the loop, then the loop is terminated, setting the iteration variable to **om**, and the **exists** test fails.

for y=f{x} **loop** ...

is allowed, and iterates through all domain elements of f, successively setting the domain element to x, and binding to y the set of all corresponding range elements paired with x.

Iterations through a sequence of integers (similar to the **do** or **for** loops of other languages) may be conveniently specified using the special form of tuple constructor for constructing tuples of integers:

for i **in** [1..100] **loop**
(statements executed 100 times with i=1,2...)
end loop;

An interesting possibility in SETL2 is to use a set former for such loops:

loop for i **in** {1..100} **do**
(statements executed 100 times)
end loop;

This loop has a similar effect to the one using a tuple former except that there is no implication as to the order in which the 100 possible values of i are selected. Good SETL2 style suggests using the set former except in cases where the order is significant, thus avoiding unnecessary overspecification.

The iterator may include a test, which selects only certain values meeting some condition to participate in the iteration:

for x **in** s | x > 5 **loop** ...
for i **in** [1,2..10] | f(i) > 0 **loop** ...

The '|' (vertical bar) symbol is read ''such that.'' The effect is to skip any values not meeting the test. For example, the following loop executes 5 times with even values of i:

for i **in** {1,2..10} | even(i) **loop** .... **end loop**; The bound variables can be any valid left hand side, and in the case of tuples, can contain dashes, as in [x,-,y].

Three other loop forms which can be written are:

while *test* **loop** -- loop while test succeeds
...
end loop;

until *test* **loop** -- loop until test fails
...
end loop;

loop -- indefinite loop
...
end loop;

The first of these forms, the **while** loop, iterates the body of the loop until the specified condition is FALSE. The test is performed at the start of the loop, so that it is possible to skip the loop execution entirely if the test condition is FALSE on initial entry to the loop.

The second form, the **until** loop, iterates the body of the loop until the specified condition is TRUE. The test is performed at the end of the loop, so the body of the loop is executed at least once, even if the condition is TRUE the first time.

The third form is an indefinite loop. The loop will continue to execute until it is terminated by executing a **stop** or **exit** statement.

10.2. Loop expressions

The **exit** statement may optionally contain an expression:

exit expression;

In this way, an entire loop may be used as a single expression, whose value is determined by the expression as evaluated when the exit statement is executed:

x := **for** s **in** set **loop**

The operator **not** yields TRUE if its operand is FALSE and FALSE if its operand is TRUE.

If an expression other than a test constructed with a test operator appears in a test context, then the value must be either TRUE (in which case the test succeeds) or FALSE (in which case the test fails). Any other value used in a test context causes an error.

10. Loops

10.1. Control loops

A loop is used where a block of statements is to be executed repeatedly until some specified condition is met, or for some specified number of times. One form of a loop in SETL2 is:

```
for iterator loop
    block
end loop;
```

This construction causes the block of statements to be executed repeatedly under control of the iterator. Within the body of the loop, any statements can be used, as well as two special statements:

```
continue;
exit;
```

Execution of the **continue** statement causes the rest of the statements in the body of the loop to be skipped, and execution continues with the next iteration (if there is one). Execution of the **exit** statement causes the current (innermost) loop to terminate.

The *iterator* has one of several forms. Three forms are:

```
x in set
x in tuple
x in string
```

In these forms, set, tuple, and string are expressions which yield values of type set, tuple, and string respecively, and x is the iteration variable, which is set to successive values from the set or tuple, or successive single-character strings in the third case. In the case of a tuple, the number of iterations is equal to the index of the last defined element, and the element values are selected in sequence. In the set case, the number of iterations is equal to the number of elements in the set and the order in which the elements are taken is arbitrary (in the same sense that **from** yields an arbitrary element). In all cases, the bound variable (x in our examples) is local to the loop, and the set, tuple, or string is evaluated on entry to the loop, and subsequent changes to the object does not affect the number of iterations nor their values.

```
for x in s loop
    print(x);                -- prints elements of s
end loop;

for x in [1,10,50] loop
    ...
end loop x;
```

The iteration variable x can actually be any valid assignment left hand side. In the case of the set iterator, this provides a convenient notation for iterating through a map:

```
for [number,root] in sqrt loop ...
```

If the set is null, then the loop body is not executed at all, and control skips immediately past the **end.**

An alternative iterator form is available, and can also be used for iterating through a map, illustrated by

```
for root=sqrt(number) loop ...
```

This is identical to the **for** loop above. However, this form can also be used to iterate through a tuple or string, assigning both the index and the element, as in

```
for y=t(i) loop     -- iterate through tuple t, assigning i and y
for c=s(i) loop     -- iterate through string s, assigning i and character c
```

Note that a character is a string that has length 1. For multivalued maps, the construct

incs	Left operand includes right as a subset (‘>= is equivalent)

Boolean Procedures

even(*i*)	Operand is even
odd(*i*)	Operand is odd
is_integer(*v*)	Operand is integer type
is_real(*v*)	Operand is real type
is_tuple(*v*)	Operand is tuple type
is_set(*v*)	Operand is set type
is_map(*v*)	Operand is map (set of pairs)
is_string(*v*)	Operand is string
is_atom(*v*)	Operand is atom
is_boolean(*v*)	Operand is boolean
is_procedure(*v*)	Operand is procedure

The equality and inequality comparisons may be used to compare values of any type for exact identity, including testing for equality with **om.** Two tuples are equal if each pair of elements in corresponding positions are equal. Two sets are equal if they have the same number of elements, and every element of the left operand is contained in the right operand.

The remaining comparisons apply only to numeric values, strings, and sets. In the string case, the ordering is like a telephone directory (e.g. "ab" is less than "aba"). The order among characters in the set depends on the implementation.

The membership tests **in** and **notin** can be used when the right argument is a tuple or set, and test for exact equality between the left operand and one of the elements of the right operand. They are also defined when both operands are strings, in which case they test whether the left side is a substring of the right side.

The operators **incs** and **subset** can only be used if both operands are sets, and can instead be denoted by <= and >= respectively. The operators are inverses of one another, i.e. a **subset** b is equivalent to b **incs** a. Likewise, < and > used with set operands test for proper subset and proper superset.

The operators **odd** and **even** can only be applied to an integer operand and test for divisibility by two.

The **is_type** operators test to see if the operand is of the specified type. Note that even though there is no type map in SETL2 (all maps are sets), the **is_map** operator is available, and tests whether the operand has the form of a map (i.e. is a set all of whose elements are pairs).

Test operators are not restricted to appearing in a context where a test is expected. If they are used in other situations (e.g. as the right hand side of an assignment), then one of two special values is yielded:

true	if the test succeeds
false	if the test fails

These values are of type boolean and are distinct from any other values. The following operators may be applied to boolean values:

Binary Boolean Operators

and	Logical and of two boolean values
or	Logical inclusive or

Unary Boolean Operators

not	Logical negation
type	Yields the string "BOOLEAN"

The operator **and** yields TRUE if both its operands are TRUE and FALSE otherwise. **Or** yields TRUE if either or both of its operands are TRUE and FALSE otherwise. **And** does not evaluate its right operand if the left operand is FALSE and similarly **or** does not evaluate its right operand if the left operand is TRUE. This means that a compound test of the following type is legal in SETL2:

if a = 0 **or** b / a > 3 **then** ...

since the division definitely will not be performed if the divisor a is zero.

end case;

This statement causes at most one of the specified blocks (sequences of statements) to be executed. The block which is executed is one whose corresponding test succeeds. The **otherwise** block, if present, is executed if all the tests fail. If more than one test succeeds, then only one of the blocks is executed, the choice of which block to execute being made in an arbitrary manner (in the same sense that the **arb** operator selects an arbitrary element from a set).

A very common use of the **case** statement involves tests of the form t=x where t is a quantity to be tested, and the various values of x are attached to statements which are to be executed if t has the specified value. A special abbreviated form of the **case** statement is available for this purpose, which has the form:

case *expression*
when *expr1* => *block1*
when *expr2* => *block2*
.
.
otherwise => *blocke*
end case;

In this case, the expression is evaluated, and then compared with each of the other expressions. The block whose associated expression matches *expression* is then executed. If no value matches then the **otherwise** block is executed. Each expression in a **when**-clause can be replaced with a comma-separated list of expressions, as in:

case *expression*
when *expr11,expr12,...* => *block1*
when *expr21,expr22,...* => *block2*
.
.
otherwise => *blocke*
end case;

In this case, the expression matches if any one of the comma-separated list of expressions matches.

In either of the **case** forms, the **otherwise** and its associated block may be omitted. If none of the branches of a **else** with no **else** block match, then execution continues with the next statement without executing any of the blocks.

9.4. Case expressions

There is also a *case expression*, which is the same as a case statement, except that the blocks are replaced by expressions, one expression per block. Note that the statements in a block must all be terminated with semicolons, whereas expressions are not terminated by semicolons. In a case expression, exactly one of the expressions is evaluated, according to which block would otherwise be executed. The value of the case expression is the same as this expression.

9.5. Boolean Values & Operators

The tests used in **if** and **case** statements and other similar constructions are usually constructed using one of the test operators. The following list indicates the condition under which the test performed by the operator succeeds:

Binary Test Operators

=	Types and values match
/=	Types or values do not match
>	Left operand greater than right
>=	Left operand greater than or equal to right
<	Left operand less than right
<=	Left operand less than or equal to right
in	Left operand is an element or sbstring of right
notin	Left operand is not an element or sbstring of right
subset	Left operand is a subset of right (‘<=’ is equivalent)

```
x.func := abs            -- Initializes func component to the absolute value proc
x.q with:= –5;           -- Appends an element to the q component of x
s := x.func(x.q(x.int))  -- Applies the procedure in the func component of
                         -- x to the component of x’s tuple in position x.int
```

Selectors can also be used to extract elements from strings, and also to specify values for procedures. That is, if the selector declaration **sel** first(1) is given, then func.first stands for func(1), where func is a procedure with an integer parameter.

9. Conditional Statements

Conditional statements allow the flow of control to be modified by the use of tests. The **if** statement is used for sequential conditional control flow, while the **case** statement should be used when the order of conditions doesn’t matter.

9.1. If Statements

The **if** statement allows one of two paths of control to be selected on the basis of a test:

if *test* **then**
    statement; *statement*;
    ...
    statement;
elseif *testx* **then**
    statement; *statement*;
    ...
else
    statement;
    ...
    statement;
end if;

The test either succeeds or fails. If it succeeds, then the sequence of statements after the **then** (called a block) is executed. If the test fails, and if there is a following **elseif** block, then the test for the next block is evaluated, and if true, then that block is executed, as if the **elseif** were a nested **if** statement. If the test is false, then succeeding **elseif** tests are evaluated. At most one block of statements is executed — as soon as a test evaluates true, and the corresponding block is executed, execution passes to the **end if** statement. The block following the **else** is executed if no other block is executed. Note that since every statement is terminated by a semicolon in SETL2, there must be a semicolon following the last statement of each block. Since the entire **if** construction is also considered to be a statement, it is also terminated by a semicolon.

The **else** and its following block are optional and may be omitted. If they are omitted and the test fails, then control passes out of the **if** which then has no effect.

9.2. If expressions

The blocks of statements in an **if** statement may be replaced by an expression — one expression per block. The expression is not terminated by a semicolon, and is evaluated if the test for the corresponding part of the **if** statement holds true. The result is an **if**-expression, whose value is the value of the expression that is evaluated. Thus

if a > 0 **then** 1 **elseif** a < 0 **then** –1 **else** 0 **end if**

is an expression which evaluates to 1, –1, or 0, according to whether a is positive, negative, or neither.

9.3. Case Statement

Another form of conditional branch is provided by the case statement, which has as one possible form:

case
when *test1* => *block1*
when *test2* => *block2*
.
.
otherwise => *blocke*

shorthand provides one method for creating multidimensional arrays. For example,

```
c := {[[0,0],"a"],[[0,1],"b"],[[1,0],"c"],[[1,1],"d"]];
x := c(0,1);                                          -- x = "b"
```

This is more attractive and more general than tuples of tuples, which is an alternative:

```
a := [ ["a","b"], ["c","d"] ];     -- Now indexing begins at 1
x := a(1)(2);                      -- x = "b"
```

Combining maps and atoms provides a way of dealing with pointers and dynamic data structures. For example, a package dealing with binary trees might use the representation T = [root, left, right, data] to represent a binary tree. To initialize a tree, we execute

```
procedure make_null_tree(rw T);
        T := [newat( ),{ },{ },{ }]; -- [root,left,right,data]
        return T(1);                 -- Returns root of empty tree T
end make_null_tree;
```

To insert data into a node x in a tree, we would simply write

```
data(x) := datum;
```

where data is the data-map in the tree (T(4)), and to create a left child of a node x in a tree T, we would execute

```
procedure make_left_child(rw T, x);  -- returns atom at the new left child
T(2)(x) := newat( );        -- T(2) is the map "left"
return T(2)(x);
end make_left_child;
```

Similarly, make_right_child and other procedures could be provided in a package for dealing with binary trees. We should also note that the use of selectors (next section) can be useful in dealing with tree structures of the form indicated here. We also note that the pair of maps left(x) and right(x) could be combined into a single map children(x) which returns a tuple [a,b] giving the nodes for the left and right child respectively. This latter representation suggests a method for generalization to ordinary trees. Also, the use of *selectors* (described below) can make the references to components of the tree structure, such as T(2), less cryptic.

8. Aggregate types

We have discussed the following datatypes, and operations on those types:

Integers	Procedures
Reals	Atoms
Strings	Tuples
Booleans	Sets

In addition, we've seen that a map is a special kind of set. What if we want another kind of type, such as an aggregate type which contains, say, a tuple, an integer, and a real? In many languages, new types can be built out of primitive types using a ''record'' structure. In SETL2, there is no need for a special syntax. Instead, aggregate types can be embedded in a tuple. Thus a tuple might contain three objects, as in [tup, i, f]. In this example, the first object, tup, is a tuple, the second object is an integer, and the third object is a procedure. Of course, it is up to the programmer to keep straight what type of object is stored where, since there is little type checking that takes place during execution. To help the programmer somewhat, there is a facility called *selectors* available for extracting elements from tuples.

A selector is a declaration (that occurs before an executible statements) that gives names to components of tuples. For example, a selector declaration might appear as:

sel q(1), int(2), func(3);

Thereafter, the notation x.q, x.int, and x.func stands for x(1), x(2), and x(3) respectively. Providing x is a tuple, then x.q extracts the first element of the tuple, which in our example, is supposed to be of type tuple. This construct allows the programmer to forget that the tuple component, which he has named q, always occurs in the first position. Note that the notation can be used on both the left and right hand sides of assignments.

```
x.q := [ ];            -- Initializes q component to [ ]
x.int := 1;            -- Initializes int component to 1
```

```
sqroot(1) := 1;
sqroot(4) := 2;
sqroot(9) := 3;
sqroot(16) := 4;
sqroot(25) := 5;
```

Reference to a non-existent element of a map (e.g. sqroot(19) in the example given) yields **om.**

Maps are a general associative device, and represent one of the fundamental data structuring devices in SETL2. For example, a structure represented as a one dimensional vector in FORTRAN is probably better treated as a map from integers in SETL2. Actually it is usually possible to represent data in a more direct way than integer indexing. For example, given a class of students, and associated test scores, rather than use two vectors based on integer indices, one containing the names and the other containing the scores, it is more convenient to represent this data as a single map from student names onto integer scores.

It is permissible to have duplicate elements in the domain, i.e. more than one pair with the same first element value. However, if the above notation is used to try to access the corresponding range element, **om** is the result, since there are multiple possibilities. However, if we use set brackets { } instead of parentheses to access the elements, the result of such a reference is to yield the set of all range values corresponding to the given domain value:

```
a := {[1,2], [1,3], [2,4], [5,5], [2,7], [2,8]};
c := a(1);                                  -- c = om
d := a{1};                                  -- d = {2,3}
e := a{5};                                  -- e = {7}
f := a(5);                                  -- f = 5
g := a{7};                                  -- g = { }
```

In general, a(x) references the range element paired with a domain element x, and **om** if there is more than one such range element, whereas a{x} returns the set of all range elements that are paired with domain element x, which will be empty if x is not in the domain of a. The reference to the set of range elements is called a multi-valued reference. This form can also be used on the left side of an assignment:

```
a := {[1,0], [1,2], [1,5], [2,5], [2,7]};
a{1} := {5,7};        -- a = {[1,5], [1,7], [2,5], [2,7]}
```

Since maps are just a special case of sets, all the operators which apply to sets (such as + for set union) can also be applied to maps. In addition the following special operators are provided for operating on maps:

Binary Map Operators

lessf	Removes pairs for one domain value

Unary Map Operators

domain	Yields domain of a map
range	Yields range of a map

Domain and **range** return the set of values of the first element or second element respectively of all pairs. The binary operator **lessf** creates a new map in which all pairs starting with a particular value are removed, and is most often used in its assigning form. All three operators cause an error if they are applied to a set which is not a map, i.e. a set which contains at least one element which is not a pair. The effect of **lessf** can also be obtained by an explicit assignment as shown in the following examples:

```
a := {[1,2], [1,3], [2,2], [2,4], [3,6], [3,7]};
b := domain a;                              -- b = {1,2,3}
c := range a;                               -- c = {2,3,4,6,7}
a lessf:= 1;                                -- removes [1,2] and [1,3]
a(2) := om;                                 -- removes [2,2] and [2,4]
a{3} := { };                                -- removes [3,6] and [3,7]
```

A special syntax is permitted when the domain elements of a map are all themselves tuples. For example, if a map takes elements of R^3 into R, then the map is a set of tuples, with each tuple having form [[x,y,z], r]. In general, the notation map(x,y,z) is a shorthand for map([x,y,z]), and map{x,y,z} is a shorthand for map{[x,y,z]}. This

```
s less:= 0;          -- s = {6,2,8}
```

Arb selects an element from the set in a non-deterministic manner. In other words, there is no way to predict which element will be selected. It may even be the case that a different value will be selected in different runs of the program, even if no changes are made to the program or data. **Arb** is thus used precisely in those cases where it does not matter which element is picked. The operator **from** picks an arbitrary element from its right operand in a similar manner, but also assigns the set with this element removed as the new value of the right operand, the picked value being assigned to the left operand. Both **arb** and **from** yield **om** if applied to a null set, and in the case of **from,** the set value is unchanged.

```
a := {1,5};
b := arb a;          -- b = 1 or 5
c from a;            -- c = 1 (or 5!)
                     -- a = {5} (or {1})
d from a;            -- d = 1 (if c was 5)
                     -- d = 5 (if c was 1)
                     -- a = { }
e from a;            -- e = om, a = { }
```

The null set is the set which contains no elements. It is written as { }, i.e. a set with zero elements listed. The # operator applied to the null set yields zero. A typical technique is to build a set value by first assigning { } to a variable, and then using **with**:= to add the desired elements to the set.

The unary operator **pow** s constructs the power set of s, which will contain $2^{\#s}$ elements. This operation should be used only for very small sets s; otherwise, execution time will be long and memory requirements large. The operator **npow** is used in conjunction with an integer argument and a set argument; s **npow** n (or n **npow** s) results in the set of all subsets of s that contain exactly n elements. Notice that **arb** and **pow** are operators, and not procedures, so that the syntax **pow** s and **pow**(s) both give the power set of s.

As with tuples, an abbreviated form is permitted for constructing sets of integers:

```
{1..10}     means   {1,2,3,4,5,6,7,8,9,10}
{3,5..11}   means   {3,5,7,9,11}
```

The general form of this construction is exactly the same as that used in the tuple case:

{*first*,*next* .. *last*}

As with tuples, the second expression indicates the direction and step size, although backwards sequences are not usually used in the set case, since the order of the elements is not meaningful in a set:

```
{1,2..10}   same set as   {10,9..1}
```

7. Maps

A map in SETL2 is a set all of whose elements are tuples of length 2 (called pairs). For example, the following assigns a map value to the identifier sqroot:

```
sqroot := {[1,1], [4,2], [9,3], [16,4]};
```

If a set has this special form, its values may be accessed using map notation:

```
x := sqroot(9);          -- x = 3
```

The actual meaning of such a map reference is to search the set for the pair whose first element matches the given domain value (9 in this case), and then the second element of this pair is yielded as the range value. Map reference notation can also be used on the left side of an assignment, the effect is to modify the value of the map appropriately:

```
sqroot(25) := 5;         -- adds the pair [25,5] to sqroot
```

The exact meaning of this assignment is to compute a new map value by first removing all pairs starting with the given domain value (there were none to remove in the above example), and then to add the specified pair. However, sqroot must either already be a map, or must be initialized as an empty set { }. Often maps are constructed by a sequence of assignment statements. For example, the map sqroot could have been constructed by the sequence:

```
sqroot := { };
```

```
q := [ ];
q with:= 5;     -- q = [5]
q with:= 7;     -- q = [5,7]
e fromb q;      -- e = 5, q = [7]
e fromb q;      -- e = 7, q = [ ]
e fromb q;      -- e = om, q = [ ] (queue empty)
```

In a similar manner, the operators **with**:= and **frome** used in conjunction allow a tuple to be used as a stack, **with**:= being the push and **frome** being the pop:

```
s := [ ];
s with:= 5;     -- s = [5]
s with:= 7      -- s = [5,7]
e frome s;      -- e = 7, s = [5]
e frome s;      -- e = 5, s = [ ]
e frome s;      -- e = om, s = [ ] (stack empty)
```

6. Sets

Finally we get to the main datatype in SETL. A set is like a tuple, except that it is unordered, and a given value can appear only once (i.e. an attempt to place a value into a set more than once is ignored). Set values are written using a similar notation to tuples, except that the set brackets are used: { and }.

```
s := {1,2,"abc"};
t := {2,1,"abc"};
u := {2,1,"abc",2};     -- s = t = u
```

The following set of operators can be applied to set operands:

Binary Set Operators

+	Set union
–	Set difference
*	Set intersection
with	Add one element to a set
less	Remove an element from a set
from	Remove an element and assign remainder
mod	Symmetric difference (exclusive or) of sets
npow	Set of subsets with fixed number of elements

Unary Set Operator

#	Number of elements as integer
arb	Select arbitrary element
pow	Power set of a set (set of all subsets)

The operators + * and – applied to sets perform standard set operations of union, intersection and difference as shown by the following examples:

```
a := {1,2,3,4};
b := {3,4,5,6};
c := a + b;     -- c = {1,2,3,4,5,6}
c := a * b;     -- c = {3,4}
c := a – b;     -- c = {1,2}
```

The operator **mod** forms the symmetric difference, so that *s* **mod** *t* is the same as the set *(s – t) + (t – s)*. The operators **with** and **less** add or remove an element from a set and are most often used in their assigning forms. **With** has no effect if the element is already present, and **less** has no effect if the element is not present:

```
s := {5,2,8};
a := s with 7;  -- s = {5,2,8}, a = {5,2,7,8}
s with:= 6;     -- s = {5,6,2,8}
s with:= 5;     -- s = {5,6,2,8}
s less:= 5;     -- s = {6,2,8}
```

A special notation is available for constructing tuples whose values consist of regular sequences of integers as shown by the following examples:

```
[1..10]      same as   [1,2,3,4,5,6,7,8,9,10]
[1,3..12]    same as   [1,3,5,7,9,11]
[2..1]       same as   [ ]
[9,8..1]     same as   [9,8,7,6,5,4,3,2,1]
[9,7..1]     same as   [9,7,5,3,1]
```

The general form of this abbreviation is:

[*first*,*next* .. *last*]

The expression *first* gives the first value generated in the sequence. The expression *next* implies both the direction of the sequence (ascending or descending) and the step between successive elements. The expression *last* gives the limit value (either the maximum for an ascending sequence, or the minimum for a descending sequence). The *next* expression, together with its preceding comma, may be omitted for ascending sequences with a step of 1. As we shall see in a later section, tuples of this type play an important role in constructing loops.

Tuples may appear on the left side of an assignment statement, providing that the right hand side is a tuple value. The effect is to perform a sequence of assignments:

```
[a,b,c] := s;      -- a = s(1), b = s(2), c = s(3)
[d,–,f] := s;      -- d = s(1), f = s(3)
[e,f] := [2,4];    -- e = 2, f = 4
[a,b] := [b,a];    -- interchange a and b
```

All the components in the tuple on the left side must be valid assignment left sides, or, as shown in the second example above, a minus sign may be used to indicate that the corresponding assignment is to be skipped.

One important point is that SETL2 treats tuples as values when it comes to assignments. Consider the following sections of code:

```
abc := 12;
cde := abc;
abc := abc + 2;     -- cde still = 12

abc := [1,2,3];
cde := abc;
abc(2) := 0;        -- cde still = [1,2,3]
```

In SETL2 the two sequences have similar effects. If you expected cde to change in the second sequence, then study it carefully. If not, then you have the correct idea already.

The operator **with** adds a single element at the end of a tuple and is most often used in its assigning form as shown in the following examples:

```
a := [1,5,10];
b := a with 6;     -- a = [1,5,10], b = [1,5,10,6]
a with:= 7;        -- a = [1,5,10,7]
```

The binary operator **fromb** removes the first element of a tuple (i.e. the element at the beginning of the tuple) and assigns it to the left operand. The right operand, which contained the original tuple operand is reassigned to contain the remainder of the tuple after removing this element. **fromb** is most often used on its own as a statement form, but it can also be used within an expression, in which case it yields the first element as its value (as well as performing the two assignments). The binary operator **frome** is similar except that it removes the element from the end of the tuple. If **fromb** or **frome** is applied to a null tuple value, then **om** is obtained and the tuple value is unchanged:

```
a := [11,26,37,17];
b fromb a;          -- b = 11, a = [26,37,17]
c fromb a;          -- c = 26, a = [37,17]
d fromb a;          -- d = 17, a = [37]
e fromb a;          -- e = 37, a = [ ]
f fromb a;          -- f = om, a = [ ]
```

The operators **with**:= and **fromb** used in conjunction allow a tuple to be used as a queue, **with**:= being the queue operation and **fromb** the dequeue operation:

```
t := [1, 9, "abc", [1, 5]];
```

As shown, the tuple can contain any values, even other tuple values, and may be of arbitrary length.

Individual elements of a tuple can be extracted by subscripting, and subscripting on the left side of an assignment allows a specified element to be modified:

```
t := [1,9,"abc","def"];
x := t(3);                -- x = "abc"
t(4) := 0;                -- t = [1,9,"abc",0]
```

Note that ordinary parentheses are used for subscripting, not tuple brackets. If a non-existent element (e.g. t(7) in the above case) is selected, then **om** is obtained as a result. It is perfectly valid to assign a value to a non-existent element, and may result in increasing the length of the tuple:

```
t := [1,9,"abc",0];
t(5) := 5;             -- t = [1,9,"abc",0,5]
```

If such an assignment creates "holes" in a tuple, the missing element values are set to undefined (i.e. to contain **om**). There is no maximum size of a tuple, although the available memory will limit the size in practice. It is possible to undefine a previously defined element by assigning **om** to it. This will either create a "hole" in the tuple value, or it will decrease the length of the tuple in the case where the last element is undefined in this manner.

The following operators are used in conjunction with tuples:

Binary Tuple Operators

+	Tuple concatenation
t*i i*t	Concatenates *i* copies (integer *i*) of the tuple *t*.
with	Appends an element to a tuple
frome	Extracts an element from the end of a tuple
fromb	Extracts an element from the beginning of a tuple

Unary Tuple Operator

#	Index of highest defined element

The concatenation operator + joins two tuples end to end to yield a new tuple as shown by the following example:

```
a := [1,2,3];
b := [6,7];
c := a + b;       -- c = [1,2,3,6,7]
```

The cardinality operator # gives the index of the highest defined element. This will be equal to the number of defined elements in the case where the tuple contains no "holes". Use of the * operator requires that one of the operands is an integer, in which case a new tuple is formed with repititions of the original tuple. The **with**, **frome**, and **fromb** operators are discussed later on.

Subtuples can be extracted using a notation similar to that used in extracting a substring:

```
a := [1,2,3,4,5,6,7,8,9];
b := a(6..8);             -- b = [6,7,8]
c := a(7..);              -- c = [7,8,9]
d := a(8..11);            -- d = [8,9]
```

These subtuple operations extend to use on the left side of assignments in the same manner as for substring assignments:

```
t := [1,2,3,4,5,6];
t(2..5) := [7,10];     -- t = [1,7,10,6]
t(3..) := [ ];         -- t = [1,7]
```

The nonstandard convention allowing negative indices also extends to tuples, so that t(–1) is the final element, and t(–3..) extracts a subtuple of the final three elements. The null tuple (which contains no elements) is written as [ ], i.e. a tuple with no elements. If the # operator is applied to a null tuple value, the result is zero. A typical technique is to create tuple values by first assigning [ ] to a variable, then executing a series of assignments which define the required element values, such as with the assignment operator **with**:=.

As a special nonstandard convention, slices with negative indices are also allowed, and extract elements from the end of the string, so that abc(−1) is the last element, and abc(−3..) denotes the last three elements.

3. Booleans, atoms, and procedure types

We have seen examples of integers, reals, and string data types. There are three other simple types: booleans, atoms and procedures. A *boolean* is a variable which has only two values, either **true** or **false**. Many tests and builtin procedures return boolean variables, and expressions can be formed using standard boolean operators, which we will discuss in Section 9.5. The literals **true** and **false** are valid boolean objects.

An *atom* is a type which is created only using the **newat** procedure:

```
v := newat( );
```

Atoms are typically used as domain elements for maps (see Section 7), and cannot be combined in arithmetic expressions.

Finally, any procedure, all of whose formal parameters are read-only (type **rd**), can be used as the value of a *procedure* variable, and can be assigned to variables, passed as parameters, used as elements in sets, maps, and tuples, and can also be executed. For example, the following bit of code assigns the builtin procedure **cos** to the variable *x*, and then uses *x* to call the function on the real value 1.0:

```
x := cos;
c := x(1.0);
```

Procedure variables may also take on values representing user-defined procedures. A procedure may be defined using a **procedure** definition, or by means of the **lambda** construction, illustrated by:

```
x := lambda(y);
  pi := 3.1415926536;
  y := 2.0*pi*y/360.;  return cos(y);
end lamda;
c := x(90.0);    -- c = 0.0
```

Procedure definitions are discussed in Section 15.3.

4. Errors & Omega

Improper operations, such as applying the / operator to string operands, normally causes termination of the program with an appropriate error message.

In addition to this class of errors, there is a special undefined state called **om** (for omega). Identifiers which have not been set to any particular value are in this undefined state, and may be thought of as containing the value **om,** although strictly speaking, **om** is not a value (but rather the absence of any value).

If an identifier contains a value (other than **om**), it may be reset to an undefined state by an assignment of the form:

```
a := om; -- a is now undefined
```

Any attempt to perform an operation on the undefined value causes an error, with two exceptions. The equality operator may be used to test whether a given value is undefined or not, as in

```
if x = om then ...
```

and the special operator ? may be used in conjunction with **om**:

```
x := expr1 ? expr2;
```

which evaluates *expr1*, and assigns its value to x, unless *expr1* evaluates to **om**, in which case x is assigned to the evaluation of expression *expr2*.

5. Tuples

A tuple is an ordered sequence of zero or more values. Tuples in SETL2 are similar to one dimensional vectors in other programming languages, but are more flexible.

A tuple may be created using the tuple brackets [and]. The following example assigns a four element tuple to the variable associated with identifier t:

Integer Procedures

even(*i*)	Test for even
odd(*i*)	Test for odd
char(*i*)	Character corresponding to ascii code
float(*i*)	Converts string to real
type(*i*)	Gives string "INTEGER"

Procedures for real variables

ceil(*r*)	Ceiling
floor(*r*)	Floor
fix(*r*)	Truncates to corresponding integer
type(*r*)	Gives string "REAL"
log,exp,cos	Various mathematical functions
sin,tan,acos,asin,atan	Other mathematical functions
atan2(*r,s*)	Operator on 2 reals to give arc tangent

Overloaded procedures

abs(*v*)	Absolute value, or character conversion
print(*v*)	Print to standard out
str(*v*)	String equivalent of the value
is_*type*(*v*)	Boolean test on type of value

The **str** procedure yields a string representation of any SETL2 data type, as would be produced by a **print** statement. Whenever **abs** has an argument that is a one-character string, then the result is the corresponding integer value of that character.

Ceil takes to a real argument to give the smallest integer which is not less than the operand, and **floor** gives the largest integer which is not greater than the operand. **Fix** converts a real to a corresponding integer value by dropping the fractional part, if any.

```
a := ceil(3.7);       -- a = 4
a := ceil(–3.7);      -- a = –3
a := floor(3.7);      -- a = 3
a := floor(–3.7);     -- a = –4
a := fix(3.7);        -- a = 3
a := fix(–3.7);       -- a = –3
```

Strings may be sliced to extract substrings. The format is:

```
string(a..b)
```

The expressions a and b are integers which select the starting and ending character in the substring. For example:

```
abc := "the quick brown fox";
cde := abc(5..8);   -- cde = "quic"
```

The following default slices are allowed:

```
abc(5)              -- same as abc(5..5)
cde := abc(5);      -- cde = "q"
abc(5..)            -- means 5 to end of string
cde := abc(5..);    -- cde = "quick brown fox"
```

These substring notations can also be used on the left side of an assignment statement. Note from the following examples that such assignments can shorten or lengthen strings:

```
abc := "hello";
abc(3..4) := "xyz";     -- abc = "hexyzo"
abc(4..) := "m";        -- abc = "hexm"
```

/	Integer division, integer result
**	Integer exponentiation
mod	Integer remainder
max	Integer maximum
min	Integer minimum

Binary Real Operators

+	Real addition
–	Real subtraction
*	Real multiplication
/	Real division
**	Real exponentiation
min	Real minimum
max	Real maximum

Binary String Operators

+	String concatenation

The meaning of an operator depends on the datatypes of the operands. For example, + means either integer addition, real addition, or string concatenation, depending on the operands. Arithmetic operators must be applied to either real or integer operands. If an operation is to be performed on one integer and one real value (for example, addition), then the **float** or **fix** procedure (or any of a number of other procedures) must be used as described later. An exception to this rule occurs with exponentiation, which allows a real value to be raised to a non-negative integer value. We thus have:

```
a := 3 + 4.0;          -- INVALID!
b := float(3) + 4.0;   -- b = 7.0
c := 3 + fix(4.0);     -- c = 7
```

Some operators and other tokens in SETL2 are written in bold letters in this document. This means that they are keywords, and are written in bold letters to distinguish them from identifiers. Any such names are reserved words, in the sense that they may not be used as identifier names. A complete list of reserved words is given in *The SETL2 Programming Language*. When the program is actually entered, the compiler ignores cases, so that both reserved words and identifier names may be entered in upper case or lower case or even in a mixture of cases.

The string concatenation operator (+) joins two strings together:

```
a := "abc";
b := "cd";
c := a + b;     -- c now contains "abccd"
```

A special set of operators, called assigning operators, not only yield a value, but also store the value into the left operand, which must be appropriate for this purpose (e.g. an identifier). There is one such operator corresponding to each binary operator. The name of an assigning operator is formed by appending the characters := to the usual operator name. For example, +:= is the addition (or concatenation) assigning operator which may be used as shown in the following example:

```
a := 3;          -- a = 3
a +:= 4;         -- a = 7
b := a +:= 1;    -- a = 8, b = 8
```

As indicated by the last line above, assigning operators can be used within an expression, as well as standing alone in a statement. Actually, the assignment operator itself is a special case of an assigning operator, and may also be used within an expression:

```
a := 3 + b := 5 + 2;        -- a = 10, b = 7
```

There are really only two unary arithmetic operators (+ and –), and the # unary operator (which returns the length of a string), although there are many arithmetic functions and other built-in procedures. Here are some of them, categorized by the type of the arguments. Thorughout the table, *i* stands for any integer or integer expression, *r* stands for a real, and *v* can be any of several different types.

```
    p := p + 1;                      -- move to next test value
    end loop;
  end printprimes;
```

This is not a particularly efficient program, and it is not even the easiest way of approaching the problem in SETL2, but it deliberately only uses a small set of simple features. The -- acts as a comment delimiter, the compiler ignores all text on a line which follows a double dash.

Now that we have seen one small program, we can study the features of the language in more detail.

2. Assignment Statements & Expressions involving integers, reals, and strings

As in most programming languages, a fundamental notion in SETL2 is that of assignment. The form of an assignment statement in SETL2 is:

identifier := *expression*;

Statements are free form and may be entered anywhere on a line, or split over line boundaries (except that a single token, such as an identifier, cannot be split over a line boundary). As indicated in the above example, every statement is terminated by a semicolon. Blanks may be used freely to separate tokens, and at least one blank or a line return must separate tokens in the case where one token ends with an alphanumeric (letter or digit), and the next one begins with an alphanumeric.

In the assignment statement, the identifier name begins with a letter, and contains letters, digits or a special break character (an underline character '_'), and is no longer than 256 characters. The case of letters is not significant. Unlike some other languages, a given identifier is not associated with a particular datatype. Instead, the type of the value in a variable associated with the identifier is determined by the last value assigned:

```
abc := 4;      -- abc now contains an integer
abc := 4.5;    -- abc now contains a real
```

The above lines contain comments which are introduced by the -- characters. All text following -- on a line is ignored. Another way of putting this is that the -- acts as an end of line signal to the compiler. Lines beginning with a -- are entirely ignored (except for being listed).

The datatypes we shall consider initially are integer, real and string types. Integer values are of unlimited range (actually limited only by the available memory). Real values have some range and precision defined by the implementation, and strings are arbitrary length sequences of characters (like most other languages, SETL2 does not exactly define the set of available characters, which depends on the implementation).

Corresponding to these three datatypes are constant denotations or literal values, which may be used wherever the corresponding value is required. The following are examples of valid constants:

```
123                  -- integer
134134145767671      -- integer
3.1415926535         -- real
0.45E+13             -- real
"abc"                -- string
"123"                -- string
"end quote\""        -- string
```

The last example shows how a double quote can be included in a string by escaping the quote mark. The null string which contains no characters is written as "". Other special characters within srings are the double backslash '\\', which stands for a single backslash, '\0', which is a zero byte (null), '\n' and '\t' for newline and tab respectively, and '\x*dd*' for a two-digit hexadecimal coded character.

Expressions are built up from constants, identifiers and operators, using parentheses for grouping in the usual manner. The following list shows the most commonly used binary operators, which compute a result from two operand values, expressed in infix form (i.e., such as in i + j or i **min** j):

Binary Integer Operators

+	Integer addition
–	Integer subtraction
*	Integer multiplication

AN INTRODUCTION TO THE SETL2 PROGRAMMING LANGUAGE

Modified from ''The SETL Programming Language,'' by
Robert B. K. Dewar
© 1979 All rights reserved

Modification prepared by
Robert Hummel

INTRODUCTION

SETL (for SET Language) is a high level general purpose language which allows a large variety of programming problems to be solved in an efficient manner with a minimum amount of effort.

This extract from a book[1] describes a portion of the SETL2 language. The reader is assumed to be familiar with some programming language and to have a basic knowledge of the principles of algorithmic programming, but no advanced knowledge of programming languages is assumed. A mathematical background at the college algebra level is sufficient. In particular, no prior knowledge of set theory is required, although the constructions of SETL do frequently resemble those notations commonly used by mathematicians using set theory.

Recently, SETL has been revised by Kirk Snyder, at NYU, in consultation with Professors Jack Schwartz and Ed Schonberg, and a new interpreter is available. There are sufficient changes that the new language is called SETL2. This document, which began as an excerpt from Chapter I of the book, has been revised, and describes a portion of the SETL2 language. This portion is sufficient to do plenty of interesting things.

The SETL book discusses various aspects of the language in detail, but is dependent on the syntax of the original SETL interpreter. There are several technical reports, written by Kirk Snyder, giving a terse but more complete discription of the SETL2 language.

ELEMENTS OF THE SETL2 LANGUAGE

This chapter gives an informal survey of most of the features of the SETL2 language.

1. An Introductory Example

The following is a small example of a complete SETL2 program. It is presented to give some perspective to the presentation which follows. It is not necessary to understand all the details of this program at this stage.

```
program printprimes;

-- This program prints out a list of prime numbers which includes all primes
-- less than a parameter value which is specified as input data.
  read(n);                        -- read input parameter
  primes := { };                  -- set of primes output so far
  p := 2;                         -- initial value to test

-- Loop to test successive values

  while p < n loop                -- loop as long as p less than n
    if not (exists t in primes | p mod t = 0) then
      print(p);                   -- no divisors, it's a prime
      primes with:= p;            -- add it to set of primes
    end if;
```

[1] *Programming with Sets: An Introduction to the SETL Programming Language,* by R.B.K. Dewar, E. Dubinsky, E. Schonberg, and J.T. Schwartz, published by Springer-Verlag, 1986.