THE SETL PROGRAMMING LANGUAGE

Robert B. K. Dewar

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 book describes the entire SETL 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.

Chapter I introduces the main features of the language in sufficient detail that many programs can be written. Remaining chapters discuss various aspects of the language in detail.

Although the description in this book is complete, it is not a precise formal definition. However, there is an appendix which contains a formal description of the permitted syntax.

CHAPTER 1

ELEMENTS OF THE SETL LANGUAGE

This chapter gives an informal survey of most of the features of the SETL language. The features described in this chapter are all included to the "SETL-S" subset of SETL. Thus it can be assumed that the features described in this chapter are available in all SETL implementations.

1.1. An Introductory Example

The following is a small example of a complete SETL 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 primes;

$ 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

 loop while p < n do               $ loop as long as p less than n
   if notexists 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;
  p := p + 1;                      $ move to next test value
  end loop;
 end program primes;
```

This is not a particularly efficient program, and it is not even the easiest way of approaching the problem in SETL, 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 dollar sign.

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

1.2. Assignment Statements & Expressions

As in most programming languages, a fundamental notion in SETL is that of assignment. The form of an assignment statement in SETL 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 is of arbitrary length, starting with a letter, and contains letters, digits or a special break character (usually an underline character). 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 $ character. All text following a $ sign on a line is ignored. Another way of putting this is that the $ sign acts as an end of line signal to the compiler. Lines with a $ in column 1 are entirely ignored (except for being listed).

The datatypes we shall consider initially are integer, real and string. 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, SETL 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
´don´´t´            $ string
```

The last example shows how a single quote can be included in a string by writing two quote marks together. The null string which contains no characters is written as ''.

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:

Binary Integer Operators

+	Integer addition
–	Integer subtraction
*	Integer multiplication
/	Integer division, real result
div	Integer division, integer result
**	Integer exponentiation
mod	Integer remainder

Binary Real Operators

+	Real addition
–	Real subtraction
*	Real multiplication
/	Real division
**	Real exponentiation

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** operators must be used as described later. The one exception to this rule occurs with exponentiation, which allows a real value to be raised to a non-negative integer value.

Some operators (such as **div**) and other tokens in SETL are written in capital letters in this book. This means that they are keywords, and are written in capital 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 Appendix nn. 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
```

Unary operators compute a value from a single operand and are written in front of the operator:

Unary Integer Operators

+	Positive (no effect)
−	Negation
str	String equivalent
float	Converts string to real
type	Gives string 'INTEGER'

Unary Real Operators

+	Positive (no effect)
−	Negation
ceil	Ceiling
floor	Floor
str	String equivalent
fix	Truncates to corresponding integer
type	Gives string 'REAL'

Unary String Operators

#	Number of characters (integer)
val	Real or integer equivalent
type	Gives string 'STRING'

The **str** operator yields a string representation of an integer or real operand. **Var** converts a numeric string into its corresponding real or integer value:

```
a := str (2 + 3);      $ a = '5'
b := val '1234';       $ b = 1234
c := str 003.000;      $ c = '3.0'
```

Ceil applied to a real gives 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 = 4.0
a := ceil –3.7;        $ a = –3.0
a := floor 3.7;        $ a = 3.0
a := floor –3.7;       $ a = –4.0
a := fix 3.7;          $ a = 3.0
a := fix –3.7;         $ a = –3.0
```

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'
```

1.3. Errors & Omega

Improper operations, such as applying the / operator to string operands, normally cause termination of the program with an appropriate error message. In some implementations, there may be a provision for optionally continuing execution following such an error, the details of such provisions are considered to be part of the implementation rather than the SETL language itself. In this book, the phrases "causes an error" and "error results" always refer to such a situation, and the details of the implementation must be consulted to determine exactly what happens.

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. The one exception to this rule is that there is a provision for using the equality operator to test whether a given value is undefined or not.

1.4. Tuples

A tuple is an ordered sequence of zero or more values. Tuples in SETL 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:

```
t := [1, 9, 'abc', [1, 5]];
```

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

In some implementations, the characters for the tuple brackets may not be available. To deal with this possibility, the alternate sequences (/ and /) can be used as replacements, so the above assignment could have been written:

```
t := (/ 1, 9, 'abc', (/ 1, 5 /) /);
```

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 can be applied to tuple operands:

Binary Tuple Operators

+	Tuple concatenation

Unary Tuple Operators

#	Index of highest defined element
type	Returns the string '**tuple**'

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".

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 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.

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 SETL 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 SETL 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:

```
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)
```

1.5. Sets

Finally we get to the datatype in SETL which gives SETL its name. 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 { and }.

```
s := {1,2,'abc'};
t := {2,1,'abc'};
u := {2,1,'abc',2};     $ s = t = u
```

In some implementations, the set bracket characters may not be available. To deal with this possibility, an alternate notation is available which uses << as the left set bracket and >> as the right set bracket, so the above examples could have been written:

```
s := <<1,2,'abc'>>;
t := <<2,1,'abc'>>;
u := <<2,1,'abc',2>>;
```

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

Unary Set Operators

#	Number of elements as integer
type	Yields the string 'SET'
arb	Select arbitrary element

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 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}
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.

As for 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}
```

1.6. Maps

A map in SETL 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. Often maps are constructed by a sequence of assignment statements. For example, the map sqroot could have been constructed by the sequence:

```
sqroot := { };
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 SETL. For example, a structure represented as a one dimensional vector in FORTRAN is probably better treated as a map from integers in SETL. 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.

For the map notation shown above, an error results if there are any duplicate elements in the domain, i.e. more than one pair with the same first element value. However, it is possible for a map to contain such a duplication, using 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);                                  $ causes an error
d := a{1};                                  $ d = {2,3}
e := a{5};                                  $ e = {7}
g := a{7};                                  $ g = { }
```

Maps which have duplicated values like this are called multi-valued maps, and the corresponding reference is 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** yield the set of values of the first element or second element respectively of all pairs. **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]
```

1.7. Conditional Statements

Conditional statements allow the flow of control to be modified by the use of tests.

1.7.1. 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;
else
    statement;
    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, then the block following the **else** is executed. Note that since every statement is terminated by a semicolon in SETL, 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 fact that **if** is considered to be a statement means that **if**'s can be nested, an **if** appearing as one of the statements in the **then** or **else** block. This nesting can be carried to any depth.

The **end if**; which terminates the **if** statement can be written in any of the following manners:

```
end;
end if;
end if tokens;
```

In the last case, tokens is one or more tokens copied from the text following the corresponding **if.** They must exactly match or an error results. The use of these identifying tokens helps to keep **end**'s straight and is particularly valuable when an **if** or similar construction is lengthy:

```
if a > b
then
    (long sequence of statements)
end if a > b;
```

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.

1.7.2. Boolean Values & Operators

The tests used in **if** 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 of right
notin	Left operand is not an element of right
subset	Left operand is a subset of right
incs	Left operand includes right

Unary Test Operators

even	Operand is even
odd	Operand is odd
is_integer	Operand is integer type
is_real	Operand is real type
is_tuple	Operand is tuple type
is_set	Operand is set type
is_map	Operand is map (set of pairs)

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 and to strings. 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** require that 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.

The operators **incs** and **subset** can only be used if both operands are sets. The operators are inverses of one another, i.e. a **subset** b is equivalent to b **incs** a.

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 SETL (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'
is_boolean	Tests whether an operand is of **boolean** type

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 SETL:

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

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

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.

1.7.3. Case Statement

Another form of conditional branch is provided by the case statement, which has the following form:

```
case of
(test1): block1
(test2): block2
else blocke
end CASE;                 $ or simply end;
```

This statement causes one of the specified blocks (sequences of statements) to be executed. The block which is executed is the one whose corresponding test succeeds. The **else** block 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 of
(const1): block1
(const2): block2
else blocke
end case;                 $ or end; or end case tokens;
```

In this case, the expression is evaluated, and then compared with each of the constants. The block whose associated constant value matches is then executed. If no value matches then the **else** block is executed.

In either of the **case** forms, the **else** 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.

More than one test or constant can be attached to a given case branch by:

```
(e1,e2,..en): block
(test1,test2,..testn): block
```

in which case the block will be executed if the expression matches any of the given values or if any of the tests succeed.

1.8. 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 SETL is:

```
loop iterator do
    block
end loop;        $ or end; or end loop tokens;
```

An alternate form (with the same meaning) is:

```
( iterator )
    block
end;        $ or end tokens;
```

In the latter case tokens, if given in the terminator, does not include the left parenthesis used to open the loop.

Either form 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;
continue tokens;
```

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). The form in which tokens are copied from the corresponding **loop** statement is useful where loops are nested, to specify which loop is being continued.

```
quit;
quit tokens;
```

This statement causes execution of the loop to be immediately terminated, and control resumes with the statement past the **end loop.** As with **continue**, the form with tokens copied from the **loop** statement may be used to indicate which loop is to be terminated when loops are nested.

The iterator itself has one of several forms. A full description of iterator forms appears in chapter 5. In this chapter we discuss the two forms which are most commonly used:

```
for x in set
for x in tuple
```

In these forms, set and tuple are expressions which yield values of type set and tuple respecively, and x is the iteration variable, which is set to successive values from the set or tuple. 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 **arb** yields an arbitrary element).

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

loop for x in [1,10,50] do
    ...
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) ...
```

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

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:

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

(for i in [1,3..99])
    (statements executed 50 times with i=1,3,5..)
end for i;
```

An interesting possibility in SETL 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 SETL style suggests using the set former except in cases where the order is significant, thus avoiding unnecessary overspecification.

Either of these iterator forms may be combined with a test which selects only certain values meeting some condition to be included in the iteration:

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

The | is read "such that" and may be replaced by the keyword **st** if the vertical bar character is not available. 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) .... **end**;

Three other loop forms which can be written are:

```
loop while test do      $ loop while test succeeds
  ...
end loop;               $ or end; or end loop tokens;

loop until test do      $ loop until test fails
  ...
end loop;               $ or end; or end loop tokens;

loop do                 $ indefinite loop
  ...
end loop;               $ or end; or end loop tokens;
```

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 **quit** statement or a **goto** which leads out of the loop.

As with the iterator form of the loop, parentheses can replace the **loop** and **do** keywords:

```
(while test)     $ loop while test succeeds
  ...
```

```
end;           $ or end tokens;

(until test)   $ loop till test fails
  ...
end;           $ or end tokens;

( )            $ indefinite loop
  ...
end;
```

1.8.1. 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, except that the keyword **for** is omitted. 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 | (or **st**) clause:

```
{a in expr | c}       $ {a : a in expr | c}
{a in [x,y..z] | c}   $ {a : a in [x,y..z] | c}
```

Notes:

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.

Error termination results from an attempt to place **om** into a set.

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
```

1.8.2. 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, except that the keyword **for** is omitted. 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 undefined, and the **exists** test fails.

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 undefined 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;
```

1.8.3. Compound Operators

Another specialized form of loop in SETL 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]
$ computes intersection over all sets in s (s is a set of sets) which contain the value 3
´´+/t               $ builds string from tuple of characters
```

1.9. Input/Output

There are three distinct types of input/output in SETL:

Stream Input/Output

Stream input/output is used primarily with "card reader" input files and "printer" output files, and 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.

Record Input/Output

Record input/output deals only with 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 intended for communication with programs written in languages other than SETL, 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 stream case.

Binary Input/Output

Binary input/output deals with SETL 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 SETL program, or some other SETL program, using binary input statements.

Full details of all three kinds of input/output are given in the chapter devoted to this purpose. In this introduction, we describe only the most frequently used input/output calls for reading standard "card reader" input, and generation of standard "printer" output.

The **read** procedure is used to read values from data lines from the standard input source in stream mode:

read(lhs,lhs,...);

This procedure call reads successive values separated by blanks or commas. 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. They may be split across line boundaries. 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) may be entered without surrounding quote marks. Such string items may not be split across line boundaries.

An * in the input stream causes the corresponding value to be set to undefined **(om).**

The values TRUE and FALSE are entered as #T and #F (or #t and #f, either upper or lower case may be used).

Set values are input using either { } or << >> set 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 [ ] or (/ /) brackets and in addition, simple parentheses are accepted as tuple brackets in input data.

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.

The special symbol **eof** (actually it is a special system function call) 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. Any input values which were not available because of encountering the end of file cause the corresponding variables to be set to undefined **(om).** The following example shows how a series of input items (up to the end of file) can be stored in a tuple:

```
input_data := [ ];
loop do
    read(next);
    if eof then quit; end;
    input_data with:= next;
end loop;
```

The **get** function may be used to read input lines one 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. The **get** function must be provided with the file name of the file to be read. A null string may be used to specify the standard input file in an implementation independant manner:

```
get('',lhs);
```

This call to **get** will cause the next input line to be read from the standard input file and assigned as a string value to lhs. **Eof** is set as described above to indicate whether or not the call attempted to read past the end of file.

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

Real values are converted either in fixed point format or exponent notation, dpending 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 surronding 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 #T and #F respectively.

Om (undefined)

An undefined value is printed as a single * (asterisk) character.

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 { }, or by << and >> if these characters are unavailable.

>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
or
print; $ Print blank line

The **eject** procedure, which uses no arguments, causes a page eject, and the **title** procedure sets a title string which is printed at the head of each page of output. In the absence of a call to the **title** procedure, the printout resulting from the **print** procedure prints continuously (over the creases) unless the **eject** procedure is used. After calling **title,** automatic page ejects are generated when the page is full (the appropriate definition of full being dependant on the environment). In titling mode, all page ejects result in printing the title together with a page number, and actual printout starts on the third line of each page. More than one call to **title** is permitted, the title being changed, but the page numbering is undisturbed. A call to **title** causes an automatic eject, so that the new title is established immediately. Calling **title** with no argument disconnects titling mode and returns to the normal (continuous printing) mode.

eject;	$ eject to new page
title(string);	$ set title to string
title;	$ disconnect titling mode

1.10. Labels and the Goto Statements

Any statement in SETL may be labeled:

```
label: statement;
```

where label is an identifier not used for any other purpose. The **goto** statement:

```
goto label;
```

causes control to pass to the labeled statement. As in any langauge which provides structured conditionals and looping statements, the use of the **goto** should be minimized. The obvious restrictions on the use of **goto**'s (e.g. against jumping into loops, or into a **case** statement from outside) apply, although it is permissible to jump out of loops and other structures.

1.11. 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.

1.12. Pass Statement

The **pass** statement:

```
pass;
```

has no effect and thus acts as a null 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 of
```

```
(1,3,5): print(i);
(2,4,7): print(i+1);
(0,6,9): pass;     $ do nothing in these cases
else print('no good');
end case;
```

1.13. Program Form

A complete program in SETL has the form:

```
program name;
(global variable declarations)
(main program block)
(procedure and operator definitions)
end program name;     $ or end; or end program;
```

1.13.1. Declarations

The global declarations, which are used if there are variables which are to be accessed directly by more than one procedure, or by the main program and some other procedure, have the form:

```
var name,name,...;
const name=value, name=value, ...;
init name:=value, name:=value, ...;
```

The **var** statement merely names a list of global variables which are assigned an initial **om** value as usual. The **const** statement defines identifiers to be associated with specified constant 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 **init** statement has is similar in form to the **const** statement, but declares variables which are initialized to the given constant value, but may be subsequently modified.

All variables and constants declared in the global declarations section may be accessed by any part of the program, including any procedures or operator definitions.

Any identifiers which are referenced in the main program block, but which are not included in the global declarations are private to the main program block and may not be accessed by procedure and operator definitions.

1.13.2. Main Program Block

The main program block consists of a series of statements followed by any refinements referenced in this series of statements. A refinement is a block of statements which is labeled with an identifier. Within the main sequence of statements, a refinement can be referenced by using its label as a statement. Refinements can themselves reference other refinements, but the definitions of refinements do not nest, they are written in a linear sequence, one after another. The following is an example of a main program block which uses refinements. Note that the definitions of the refinements themselves are similar in form to normal label definitions, except that the name is followed by two colons.

```
program quadratic;

input_data;
solve_equation;
output_results;

input_data::
      read(a,b,c);
      print(a,b,c);
      check_eof;

solve_equation::
      x := (-b + sqrt(b:b-4:a:c)) / (2*a);

output_results::
      print(' root is ', x);

check_eof::
      if eor then print('improper data'); stop; end if;

end program quadratic;
```

Refinements are executed by inserting the series of statements of the refinement in place of the reference to the refinement. Unlike procedures, there is no way of passing parameters, and the statements of the refinement have full access to all identifiers of the main program, including its labels. A given refinement may be referenced only once. If a section of code is to be used more than once, it should be made a procedure rather than a refinement.

1.13.3. Procedure Definitions

The form of a procedure definition is:

```
proc prname(arg1, arg2, arg3);
      (local declarations)
      block
      (refinements)
end proc tokens; $ or end proc; or end;
```

This defines a procedure whose name is prname, which must be an identifier not used for any other purpose in the program. 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 globally declared identifiers. Within the procedure, these names act as ordinary identifiers. 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 **init** statements in the same format as that used in the global declaration section. Any names declared in this manner are strictly local to the procedure, and must be different from any names declared as global. Identifiers used in the body of the procedure which are not declared either locally or globally are taken to be local, as though they had been declared locally using a **var** statement. The initialization of variables to **om** (if declared by **var,** or implicitly declared) or to the specified values in the **init** statements occurs on each entry to the procedure.

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 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);
proc prname(arg1,arg2);
return [10,20,a+b];
end proc;
```

Procedures in SETL 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:

```
proc factorial(arg);
    if arg=1 then
        return 1;
    else
        return arg * factorial(arg-1);
    end if;
end proc factorial;
```

Procedures may use refinements in the same manner as described for the main program block. These refinements are private to the procedure, and their definitions occur prior to the **end proc;** which terminates the procedure definition. There is no nesting of procedure definitions, all procedures can be called from anywhere in the program.

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:

> **proc** 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.

There is also a form which allows a variable number of arguments to be passed to a procedure:

> **proc** name (type arg1, type arg2, .. type argn (*));

Such a procedure may be called with any number of arguments greater than n-1. The extra arguments are gathered into a tuple which is assigned as the value of the argn parameter. Thus the reference argn(2) in the body of the procedure refers to the n+1'th argument in the call. The special symbol **nargs** gives the total number of arguments present in the call. Only the last parameter may be followed by (*) to indicate that it is variable in length. A variable length parameter may as usual be specified to be **rd, rw** or **wr** (with **rd** being the default). In the **wr** case, the initial value of argn is the null tuple value.

1.13.4. Operator Definitions

A special form of procedure definition is used to introduce a program defined operator. It is identical to an ordinary procedure except for the initial definition line which replaces the **proc** line:

```
op .name(a);      $ to define a unary operator
op .name(a,b);    $ to define a binary operator
```

The effect of such a definition is similar to that of the corresponding procedure declaration, except the call uses ordinary operator (expression) format. The operator name is preceded by a period, both in the definition (as shown above) and when the operator is used in an expression.

The precedence of all binary operators defined in this manner is lower than that of the standard system operators except for the assigning operators. Operators defined in this manner always have read only arguments (as though **rd** were specified). In the case of a defined binary operator, the corresponding assignment operator and compound operator forms are automatically made available.

CHAPTER 2

PROGRAMMING EXAMPLES

At this stage we have seen enough of the SETL language to show some complete programs. All programming languages have an associated style of programming. The purpose of this chapter is to give an idea of typical SETL programming style.

In each case, the statement of the problem is given first (in quotations). Following this, the program is developed in a "stream of conciousness" fashion, explaining the steps in obtaining the solution, and giving fragments of the program as they are created. Finally, the entire program is given.

2.1. A Curriculum Planning Problem

"In planning the sequence of presentation of topics in a course, one objective is to present topics in an order which ensures that all necessary prerequisites for understanding one topic have been covered before it is presented." (The same problem arises in writing a book -- hopefully this book is an example of a solution to this problem!)

"The purpose of the program is to compute a possible sequence of topics given data on prerequisites. The input data will be pairs of topics, each of which is identified by a string name. For example, one input line might be:

```
'prime numbers' , 'greatest common divisor'
```

which means that "prime numbers" must be presented before "greatest common divisor" is considered. In general it will be necessary to trace back chains of requirements. If there is also an input line:

```
'division' , 'prime numbers'
```

then of course it will be necessary to present division before discussing the greatest common divisor problem.

"For a given set of data, more than one order may be possible. For example, suppose the input data is as follows:

```
'a' , 'b'
'b' , 'c'
'd' , 'c'
```

then three possible orderings exist:

```
'a' , 'b' , 'd' , 'c'
'a' , 'd' , 'b' , 'c'
'd' , 'a' , 'b' , 'c'
```

For this problem, the program is only required to print out one (arbitrarily chosen) possibility where more than one exists.

"Another possible situation is that no acceptable ordering exists as in the following example:

```
'a' , 'b'
'b' , 'c'
'c' , 'a'
```

where it is obviously impossible to satisfy the required condition. If this situation is encountered, the program is expected to print an error message."

In approaching a problem of this complexity, the first decision to be made is the form in which data is to be represented inside the program. If this decision is made incorrectly, then trouble will be encountered in constructing the algorithm. For example, the decision to store the pairs of prerequisite information in a **tuple** would be an error in this case, as will become clear later on. The general rule in SETL is to use maps wherever possible. This may take some practice, especially if you are used to programming in some other language, but remember this simple principle: find the maps, they are always there!

In the problem at hand, we can naturally represent the pairs of data as a map. This map will be multi-valued, since a given topic may be a prerequisite for more than one other topic. The following example (for a course on wine-making) will be used throughout the remaining discussion:

```
'grapevines' , 'harvest'
'hiring' , 'harvest'
'harvest' , 'fermentation'
'yeast' , 'fermentation'
'alcohol' , 'yeast'
'bottles' , 'bottling'
'hiring' , 'bottling'
'fermentation' , 'bottling'
'bottling' , 'marketing'
'hiring' , 'marketing'
```

In addition to the basic map which contains the prerequisite data, it will be useful to build an auxiliary set which contains the list of topics. The following initial code for the program will read in the data and build these structures:

```
program course;
follows := topics := { };
loop do
      read(a,b);
      if eof then quit; end;
      follows with:= [a,b];
      topics with:= a;
      topics with:= b;
end;
```

Note that since topics is a set, it cannot contain duplicate elements, so each topic occurs only once in topics, even if it occurs more than once in the input data.

As the initial step we can output any topic which has no prerequisites. In terms of our data structure we are looking for an element in topics which has the property that there is no other element in topics which is its prerequisite. In SETL this can be expressed by:

```
if exists next in topics
            | not exists a in topics
                        | [a,next] in follows
```

After evaluating this test, next will be a possible choice for the first topic to be output. If there is more than one topic which meets the requirement, then one of them is arbitrarily chosen. In the example being considered, a possible selection for next would be 'bottles', which has no prerequisites.

We can build an algorithm using this approach by successively removing elements from topics. The above test can then be used to extract topics in sequence which meet the requirement that they do not have prerequisites among those topics not yet chosen. The test will finally fail when there are no topics which meet the requirement. If this happens because topics is null (i.e. all topics have been output), then all is well, otherwise there is a "cycle" in the data which means that it is impossible to find an acceptable order. This approach results in the following completion of the program:

```
loop while exists next in topics
            | not exists a in topics
                        | [a,next] in follows
do
      topics less:= next;
      print(next);
end loop;

if topics /= { }
then
      print('No ordering is possible');
end if;

end program course;
```

The second time through the loop, assuming 'bottles' was picked on the first loop, 'bottling' might be picked. Although the pair ['bottles','bottling'] is in follows, it does not prevent the choice, since 'bottles' was removed from topics in the first loop.

This program is complete but inefficient, in that each time one element is removed from topics, the entire search process is repeated. Except for skipping the one element which was just output, the calculations involved in this search are unchanged. This means that we are repeating work unnecessarily.

This is a typical situation which arises in the construction of efficient algorithms. What we need is some way of "remembering" the results of the previous computation (in this case the search). After an element is removed, we just need to modify the remembered result to reflect the change which has occurred.

In this particular example, the key is to build an auxiliary map which shows the number of prerequisites of each topic, counting only topics which have not yet been output. At each stage, looking for a topic with no prerequisites means finding a topic whose number of prerequisites is zero, as determined by reference to the auxiliary map. When this element is removed, the auxiliary map is adjusted to reflect the change, which can be done without recalculating the entire map. To implement this idea, we must first build the auxiliary map:

```
numpre := {[a,0] : a in topics};
(for [a,b] in follows) numpre(b) +:= 1; end;
```

For the example at hand, the numpre map would contain:

```
{   ['grapevines'     , 0],
    ['harvest'        , 2],
    ['hiring'         , 0],
    ['fermentation'   , 2],
    ['yeast'          , 1],
    ['alcohol'        , 0],
    ['bottles'        , 0],
    ['bottling'       , 3],
    ['marketing'      , 2]    }
```

Now the main loop of the algorithm appears as follows:

```
loop while exists next in topics
                    | numpre(next) = 0
do
    topics less:= next;
    print(next);
    (for a in follows{next})
        numpre(a) -:= 1;
    end;
end loop;
```

The test for cycles is unchanged. For the current example, a possible choice for next in the first loop is 'hiring', since numpre('hiring') is zero. In the adjustment loop following this choice, follows{'hiring'} is the set of topics which had 'hiring' as a prerequisite, i.e. the set:

{'harvest' , 'bottling' , 'marketing'}

Since 'hiring' is to be removed from topics, the number of predecessors of each of these elements should be reduced by one to give:

numpre('harvest') = 1
numpre('bottling') = 2
numpre('marketing') = 1

The remainder of the numpre map is unchanged. On the next loop, we might get 'grapevines' since numpre('grapevines') is zero. Now follows{'grapevines'} is the singleton set {'harvest'} so we make one change to the numpre map:

numpre('harvest') = 0

making 'harvest' an eligible possibility for selection on the third loop.

This is a more efficient procedure than our previous attempt. However, we still have a source of inefficiency, since numpre is searched at each step to find zero entries. Applying the same principle of avoiding doing the same thing twice, we can make one final improvement in the program. To avoid the search of numpre, we maintain an auxiliary set, called nopre, which is the set of topics whose numpre value is zero. At each stage we need merely pick an element from nopre. As the numpre map is adjusted when the element is removed, we check for any entries whose numpre value becomes zero, and add them to nopre. This avoids all searching. The first step in following this approach is to initialize the nopre set right after the initialization of numpre:

nopre := {a **in** topics | numpre(a) = 0};

For the example we are using, nopre would have the initial value:

{'alcohol', 'hiring', 'grapevines', 'bottles'}

Remember that the order of elements in a set is not defined, so that when an element is picked from nopre, there is a four way uncertainty. This aspect of the SETL program models the fact that there is more than one acceptable solution to the problem. The main loop of the program now becomes:

```
loop while nopre /= { } do
    next from nopre;
    topics less:= next;
    print(next);
    (for a in follows{next})
        numpre(a) -:= 1;
```

```
            if numpre(a)=0 then nopre with:= a; end;
        end;
    end loop;
```

If 'hiring' is picked in the first loop, then numpre is adjusted as described above and nopre becomes:

{'alcohol' , 'grapevines' , 'bottles'}

In the second loop, suppose that 'grapevines' is selected. As numpre('harvest') is decremented, it becomes zero, so 'harvest' is added to nopre, giving the new nopre value:

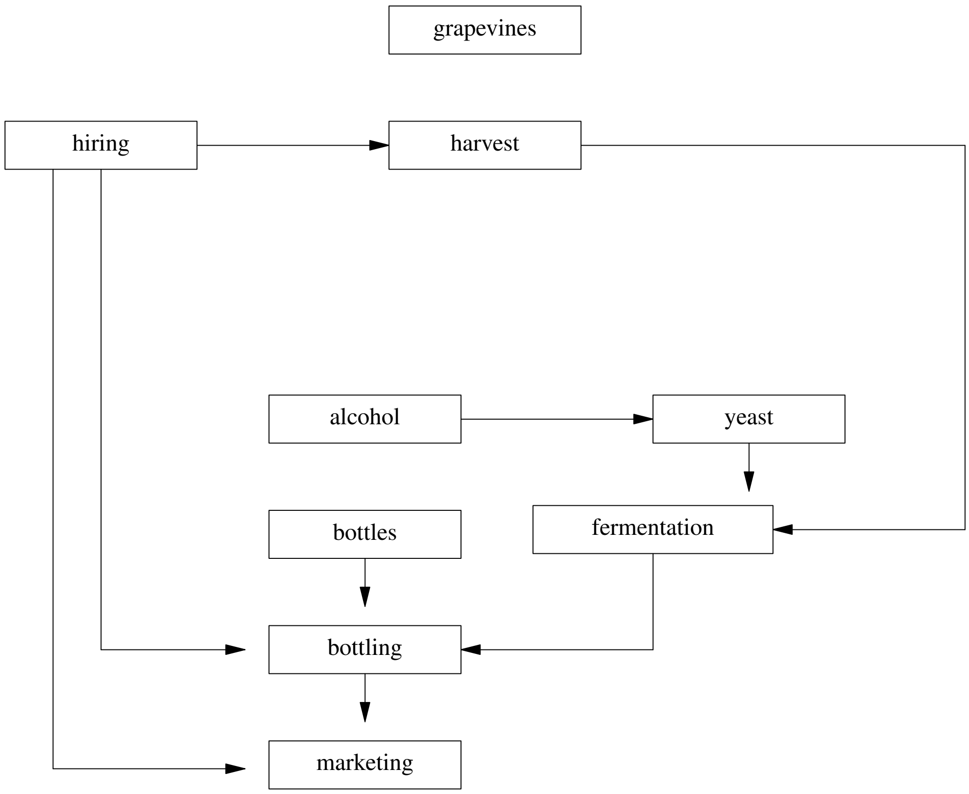
{'harvest' , 'alcohol' , 'bottles'}

We now give the complete program with comments added. This final version contains two additional refinements. The input data is printed out, which is a good general procedure to follow and makes the program easier to use. The second change is a change in the test for cycles which avoids the need to remove elements from topics in the loop.

```
program course;

$ follows is map showing prerequisites
$ topics is set of all topics

follows := topics := { };

$ loop to read in data, building follows, topics

loop do
    read(a,b);
    if eof then quit; end;
    follows with:= [a,b];
    topics with:= a;
    topics with:= b;
    print(a,'  ',b);
end loop;

$ build auxiliary information structures
$ numpre(x) = count of remaining prerequisites of x
$ nopre is set of topics with numpre(x) = 0

print( );
numpre := {[a,0] : a in topics};
(for [a,b] in follows) numpre(b)+:=1; end;
nopre := {a in topics | numpre(a)=0};
```

```
$ loop to print elements in appropriate order
$ adjusting numpre and nopre for element picked

loop while nopre /= { } do
    next from nopre;
    print(next);
    (for a in follows{next})
        numpre(a) -:= 1;
        if numpre(a)=0 then nopre with:= a; end;
    end for;
end loop;

$ test for cycles, indicated by a remaining
$ topic with numpre non-zero

if exists a in topics | numpre(a) > 0 then
    print('No ordering is possible');
end;

end program course;
```

This problem is actually a specific example of a general problem known as "topological sorting", in which a linear order must be generated from a partial order. A partial order is essentially a directed graph. In fact a multi-valued map, such as we have used in this program, is often a convenient representation for a directed graph. For the data used in the example, the map "follows" may be thought of as representing the following graph:


grapevines
hiring
harvest
alcohol
yeast
bottles
fermentation
bottling
marketing

CHAPTER 3

CHARACTER SET & DATATYPES

This chapter describes the most elemental aspects of the SETL language: how characters can grouped together to form those symbols which are the main consituent components of a SETL program.

3.1. Character Set

SETL programs are written using characters from the standard SETL character set which contains the following characters:

A-Z	upper case letters
0-9	numeric digits
<	less than
>	greater than
(	left parenthesis
)	right parenthesis
´	quote
.	period
,	comma
:	colon
;	semicolon
/	slash
+	plus
–	minus
$	dollar sign
?	interrogation
#	number sign
_	underline
{	left set bracket
}	right set bracket

[	left tuple bracket
]	right tuple bracket
|	such that character

When using SETL on a particular machine, it may be the case that certain of these characters are not available. For the last five characters in the list, standard substitutes are available (since these are the characters which are most likely to be missing):

{ can be written as <<
} can be written as >>
[can be written as (/
] can be written as /)
| can be written as ST

The remaining characters are always available (at worst the graphic used will vary from the one printed in the above list). This means that the process of translating a SETL program written in the standard character set into the required character set for actually running the program is at most a one to one substitution.

Certain implementations may augment this character set. In particular, if two cases of letters are available, then lower case letters are permitted. SETL treats upper and lower case letters as indistinguishable (except in string constants as further described below).

In addition to this standard character set, there is also defined a publication character set which is fully described in appendix nn. The publication set contains various "mathematical" symbols and is chosen without regard to availability of characters on actual computers. Nevertheless, on many computers it will be possible to implement some of these characters, or at least to supply acceptable substitutes for them. A particular implementation will describe which if any of the characters in the publication set it implements. A program may use such characters as are defined to be implemented, but if the program is moved to another implementation which does not implement the same subset of publication characters, then a translation problem may be encountered. Therefore the general rule is to use only the standard character set if there is any possibility that the program will be moved to another implementation.

The above discussion of character sets applies in all contexts except for input data and string constants appearing in the program. In both these cases, the full character set available on the particular computer in use are used. Since such character sets vary widely from one machine to another, SETL makes no attempt to define the set of available characters for these uses. Note also that upper and lower case letters are distinct in these contexts. Programs which are intended to be moved from one implementation to another can minimize conversion difficulties by restricting data and string constants to the standard character set.

3.2. Syntactical Tokens

A program is a stream of syntactical tokens which are in one of the following categories:

Identifier

A string of letters, digits and a special character called the break character (in this book, and in most implementations, the break character is an underline). An identifier must start with a letter and contain no blanks. It can be any length, but must not be split across a line boundary. Upper and lower case letters can be used interchangably and are equivalent.

Examples: thisisalongidentifier
this_is_a_longer_identifier
i
j107x

None of these identifier names can be the same as any reserved keyword or operator name. Appendix nn contains a complete list of such reserved names.

Keyword

A keyword is a string of letters which is used for a special purpose such as introducing a control structure. All keyword names are reserved words and may not be used as identifiers. Appendix nn contains a complete list of standard keywords in SETL.

Punctuation

Certain special characters and sequences of special characters appear as separators:

; semicolon
(left parenthesis
) right parenthesis
[left tuple bracket
] right tuple bracket
(/ alternate left tuple bracket
/) alternate right tuple bracket
{ left set bracket
} right set bracket
<< alternate left set bracket
>> alternate right set bracket
: colon
| such that
, comma
.. integer range
... alternate for integer range

Operators

Certain special characters and sequences of special characters are used for names of standard operators:

:= assignment, assigning operator suffix
+ addition, set union, concatenation
− subtraction, set difference
* multiplication, set intersection
/ division, compound operator suffix
** exponentiation

<	less than
>	greater than
<=	less than or equals
>=	greater than or equals
=	equals
/=	not equals
#	cardinality
?	interrogation

Other standard operators have names which are similar in form to identifier names. All such names are reserved and cannot be used as identifier names in a program. Appendix nn contains a complete list of such reserved names.

User Defined Operators

User defined operators have names which start with a period. The names themselves may match other identifier names, or reserved words, although such multiple usage is usually not desirable. A given operator name may be used for only one operator.

Denotations

Constant values of basic datatypes are represented by tokens called denotations (e.g. 123 is a token representing an integer value). A subsequent section gives the rules for constructing denotation tokens.

Comments

Normally the SETL compiler processes all columns of each input line. However, the character $ (dollar) is treated as an end of line signal and any text following the $ is ignored (apart from being listed in the program listing generated by the compiler). This means that arbitrary comment text may be written on any line following a $ sign. In particular, lines which have a $ in column one are entirely ignored and thus function as comment lines. The SETL compiler always ignores blank lines, so they may be used freely to space the program and comment text.

3.3. Datatypes

The following primitive datatypes appear in SETL:

integer

Integers are signed integer values. Unlike many other programming languages, SETL places no limit on the magnitude of integers, although enormous integers might eventually exceed the available storage. Of course the programmer should expect that adding two one digit numbers will be more efficient than adding two ten thousand digit numbers. There is only one zero in SETL and it is considered to be neither positive nor negative.

real Real number values correspond to those available on the machine in use. Typically this will mean that implementation dependant limits and accuracy will apply. SETL has only one precision of real numbers, and will choose to use "double precision" values on machines whose word size is small (e.g. 32 bits). There is only one real zero value in SETL and it is considered to be neither positive nor negative.

string
These are arbitrary length strings of characters. As in other languages, SETL is vague about what set of characters will be permitted in STRING values. In practice, the allowed set will correspond to the characters available on the particular machine in use.

boolean
There are two boolean values, TRUE and FALSE. These values are yielded as the result of test operators and are often used to control the flow of execution.

atom These are special unique values which are used in constructing data structuring maps. The use of atoms is described in the chapter on data structures.

om **Om** is used to represent an undefined value in the following circumstances:

Undefined variable
Undefined element of a tuple
Value of map at undefined point
Element selected from the null set
Compound operator applied to null set or tuple

Om is not a value from a technical point of view and does not have a type (the **type** operator applied to **om** causes an error), however it is sometimes convenient to regard **om** as being the "value" of an undefined variable or element.

3.4. Denotations

For each of the basic datatypes, denotation tokens can be constructed which represent constant values of particular datatypes.

3.4.1. Integer Denotations

An integer denotation consists of a sequence of digits of any length and has as its value the corresponding decimal number. No blanks may appear within the constant and a single constant must appear entirely on one input line.

There are no negative denotations as such, if −123 appears in a program, then it is composed of two tokens: the − sign is a unary operator, and the 123 is an unsigned integer denotation.

Examples:

```
0
13
0013              $ same value as 13
12349769762347612497673467124923761246767676712497614
```

3.4.2. Real Denotations

A real denotation consist of an optional integer part consisting of a string of digits, followed by a period (decimal point), followed by a non-empty string of digits which is the fractional part, followed by an optional exponent.

An exponent consists of the letter E (upper or lower case if both are available), followed by an optional sign (+ or −), followed by a non-empty string of digits.

Blanks may not be used within the denotation, and a single denotation must be contained entirely on one line of the program.

Examples:

```
3.141592653589793
0.0
 .0                    $ but not 0.
 .01e+10
 .01e-10
1.0E0
1.0E+10                $ but not 1.E10 or 1E10
```

3.4.3. String Denotations

String denotations consist of a series of zero or more characters enclosed in apostrophe characters ("single quotes"). If blanks appear in this sequence, they are significant, and represent blank character values.

Any of the characters available on the machine in use may appear in a **string** value, although programs which are intended to be run on a variety of different computers should restrict their use to commonly available characters (e.g. those used within the SETL language itself) to avoid translation problems.

If the string quote mark itself appears within a **string** value, then it must be represented as a sequence of two successive string quotes with no intervening blank.

If the sequence of characters crosses a line boundary, then a string quote must signal the end of the characters on the first line, and another quote signals the start of the characters on the second line. This sequence of quote, line break, quote is called a string break and is not included in the actual string value.

Examples

```
'Don''t tread on the grass'
'a m ,;;;;;'
''                  $ the null character string
```

3.4.4. Boolean Denotations

The denotations TRUE and FALSE can be used to stand for the two possible boolean values.

3.4.5. Other Denotations

The symbol **om** may be used to represent the undefined value (it is not strictly a denotation, since **om** is not strictly a value). The only sensible contexts for the appearence of **om** are the right hand side of an assignment, and as an operand for an = or /= test whose purpose is to test for undefined.

There are no denotations for **atom** values.

CHAPTER 4

EXPRESSIONS & ASSIGNMENT STATEMENTS

An expression in SETL is used to compute a value or test some condition. There are five forms for an expression:

- Basic operand, includes identifiers, denotations, set and tuple formers, subscripted references to tuples, map references, conditional expressions and parenthesized expressions.
- Special system value (**eof, newat, nargs, ok, lev, time, date**).
- Unary operator, which is written in front of its operand which is itself an expression and which causes a computation to be made using the operand value as input.
- Binary operator, which is written in between its two operands, which are themselves expressions. Causes a computation to be made using the two operand values as input.
- Quantified test, one of three forms **(exists, notexists, forall)** which excute an implied loop testing a condition and yield a boolean result.

These rules leave some ambiguity in forming an expression, for example in the expression a+b*c, is the operator + applied to the two operands a and b*c, or is the operator * applied to the two operands a+b and c? These ambiguities are resolved by the precedence rules described later in this chapter, for example these rules specify that the first alternative is the one taken in this case. It is also possible to use parentheses to alter or emphasize the order prescribed by these rules. For example:

```
1 + 2 * 3       $ = 7 by precedence rules
1 + (2 * 3)     $ = 7
(1 + 2) * 3     $ = 9
```

The following sections give the exact rules for forming basic operands, together with a list of all the standardly defined operators.

4.1. Basic Operands and Special System Values

This section describes the various possibilities for basic operands in expressions.

Denotations

Denotations stand for the value which they denote. For example the denotation 123 stands for the integer value 123.

Identifiers

Identifiers stand for the value contained in the associated variable, as set by a previous assignment statement.

Tuple Enumeration

A tuple value constructed by enumeration consists of zero or more arbitrary expressions, separated by commas, with values other than **om** and enclosed in tuple brackets [and].

Integer tuple former

An integer tuple former has one of the two forms:

```
[ expression1, expression2 .. expression3 ]
[ expression1 .. expression3]
```

The effect is to construct a tuple containing the specified range of integer values. The starting value is given by expression1. If expression2 is present, then it specifies both the step size and direction of the sequence which produces the remaining values. The step is the difference expression2 minus expression1. This value must be non-zero. If it is positive, then expression3 is the maximum value which terminates the sequence, if negative, then it is the minimum value terminating a reverse sequence. If expression2 is omitted, then the default step size is 1 and the sequence is ascending (i.e. the default for expression2 is expression1 plus 1).

Tuple Former

A tuple former is written:

```
[ expression : iterator ]
```

and computes a tuple value in an implied loop. The syntax and meaning of tuple formers is further described in the section on loops.

Set Enumeration

A set value constructed by enumeration consists of zero or more arbitrary computational expressions with value other than **om**, separated by commas, and enclosed in set brackets { and }.

Set Former

A set former is written

```
{expression : iterator}
```

and computes a set value in an implied loop. The syntax and meaning of set formers is further described in the section on loops.

Integer set former

An integer set former has one of two possible forms:

```
{ expression1, expression2 .. expression3 }
```

{ expression1 .. expression3 }

The effect is to construct a set using the sequence of integers specified. This sequence of integers is the same as that implied in an integer tuple former as previously described, except that the resulting set does not retain the order of the elements, since sets are always unordered.

String Slice

A string slice obtains a substring from a **string** value. The syntax is:

```
string(start..end)
```

where string is the string value, which must be a basic operand, and start and length are arbitrary expressions yielding integer results. Start is the starting position (with the first character in the string numbered 1), and end is the ending position numbered in the same manner. The resulting string is the substring which includes both the starting and ending characters. Both the start and end values must be in range (greater than zero and less than or equal to the length of the string). The one exception to this rule occurs when the end value is less than the start value. In this case the start and end values need not be in range and the result is always a null string. The following abbreviated forms are permitted:

```
string(start)      $ means string(start..start)
string(start..)    $ means string(start..#string)
string(..start)    $ means string(1..start)
```

Tuple Slices

Tuple slices have the same basic format as string slices:

```
tuple(start..end)
```

The value tuple, which must be a basic operand, is the tuple from which a slice is to be selected. Start and end are arbitrary integer expressions which specify the desired subtuple. Both start and end must be in range (greater than zero and less than or equal to the length of the tuple). As in the case of string slices, the one exception occurs if the end value is less than the start value, in which case a null tuple is obtained as the result regardless of whether or not the start and end values are in range. Note that a tuple can never have any undefined values at the end, so the resulting tuple is shortened if necessary to meet this requirement as illustrated by the following examples:

```
a := [1,3,7,9];
a(8) := 13;
b := a(2..3);        $ b = [3,7]
b := a(2..4);        $ b = [3,7,9]
```

```
b := a(2..5);      $ b = [3,7,9]
b := a(11..10);    $ b = [ ]
```

The following short hand notations are available:

```
tuple(start..)   $ means tuple(start..#start)
tuple(..start)   $ means tuple(1..start)
```

Note that, in contrast to string slicing notation, the form tuple(start) is a selection, not a one element slice.

Tuple Selection

A tuple selection, or tuple subscripting operation yields a specified single element from a tuple. The form is:

```
tuple(index)
```

where tuple is a basic operand yielding the tuple value to be subscripted, and index is an arbitrary expression with a positive value which selects the desired element. If index exceeds the maximum index, or corresponds to an undefined element value, the result is undefined **(om).** An error results if the index value is zero or negative.

Single Valued Map Reference

A single valued map reference is written:

```
map(domval)
```

where map is a basic operand which yields a set value which must contain only pairs (two element tuples), and domval is an arbitrary expression giving the domain value. Domval must not be **om,** and there must be exactly one pair [x,y] in map such that domval=x. The result of the map reference is the range value y.

Multi-valued Map Reference

A multi-valued map reference is written:

```
map{domval}
```

where map is a basic operand or parenthesized expression which yields a set value which must contain only pairs (two element tuples), and domval is an arbitrary expression giving the domain value. The result is expressed by the following equation:

map{domval} = {y : [x,y] **in** map | x = domval}

In particular, the value is { } if there is no pair in the set with a matching domain value. If there is only one pair, the result is a singleton set containing the one range value.

Multi-argument Map References

It is sometimes convenient to have a multi-argument map, i.e. a map whose elements are retrieved using two or more subscript values. Such maps are modeled in SETL by using a map where the elements of the domain are themselves tuples of values. For example, the equivalent of a 2 by 2 array can be modeled as the map:

```
{   [[1,1] , v11],
    [[1,2] , v12],
    [[2,1] , v21],
    [[2,2] , v22]   }
```

A special notation is available to facilitate references to such a map:

```
map(a,b..n)   means   map([a,b...n])
map{a,b..n}   means   map{[a,b...n]}
```

This means that the extra tuple brackets can be omitted in such a reference, giving a notation similar to multi-dimensional array references in other languages.

Function Calls

A function call is a call to a procedure which returns a value, and has the general form:

```
fname(expression,expression,...expression)
```

Further details on function calls are contained in the chapter on procedures.

Special Value

Certain special reserved names correspond to values available from the SETL run time system. The following is a list of all such names:

newat	$ yields a unique atom
eof	$ tests for end of file
nargs	$ number of arguments
lev	$ current backtracking level
ok	$ backtracking environment switch
time	$ time of day
date	$ current date

Newat yields a new atom value which has the property of being different from any previously obtained atom. The use of atoms in general, and the **newat** function in particular, is further described in the chapter on data structures.

Eof is used following a read or other input function call. It returns a boolean value (TRUE or FALSE) indicating whether the read just performed caused an end of file to be encountered (TRUE means that an end of file was encountered). If **eof** is referenced before any input operation has been performed, the result is FALSE.

Nargs yields the number of arguments which appear in the call to a procedure. It is used within the procedure, and is most useful in conjunction with the feature which allows a variable number of arguments to a procedure. If **nargs** is used outside any procedure (in the "main" program), the value is zero.

Lev and **ok** are only used in conjunction with the backtracking feature, which is fully described in a separate chapter.

Parenthesized Expression

Any expression enclosed in parentheses can be used as a basic operand. This is the rule which allows the use of parentheses to control the order of operations.

Conditional Expressions

Conditional expressions (**if** expressions and **case** expressions) may be used as basic operands. These constructions are fully explained in the following chapter.

expr *Blocks*

The **expr** block allows a value to be computed from a sequence of statements, and used as a basic operand in any suitable context, e.g. as the operand of some other operator. The form is:

expr block **end**

The effect is to execute the sequence of statements in block. At least one of these statements will be, or contain, a **yield** statement:

yield expression;

As soon as the **yield** statement is encountered, execution of the **expr** block is terminated, and the value of the **expr** block is the value of the expression in the **yield** statement. It is possible to have more than one **yield** **yield** statement in the same **expr** block, and the last statement is not required to be a **yield** statement, although it often will be. If execution of the **expr** block completes without executing a **yield** statement, then the result is **om.** It is possible, though unusual, to nest **expr** blocks, i.e. to use an **expr** block as an operand within another **expr** block, but in this case, a **yield** statement in the inner block always refers to the inner block. The use of a **yield** statement other that statically inside an **expr** block is erroneous, in particular, it is not valid to execute the **yield** in a function or refinement called within the **expr** block.

4.2. Operators

Operators yield a value using as input one (unary) or two (binary) operands. The following is a complete list of predefined operators in the SETL system. The effect of each operator is defined informally in this section. A more formal set of definitions appears in appendix nn where each operator is defined precisely using a SETL operator definition. This appendix may be consulted to determine the exact effect of the operator in special cases.

4.2.1. Unary Operators

Unary operators compute a value from a single input operand which is written to the right of the operator token. In the following list, the allowable operand datatype is indicated. If an operator is used with a datatype which is not in the list, then an error results.

+ integer	The result is the integer operand, unchanged in value.
+ real	The result is the real operand, unchanged in value.
– integer	The result is the negative of the integer operand. –0 is equal to 0.
– real	The result is the negative of the real operand. –0.0 is equal to 0.0.
# set	Number of elements in the set as an integer.
# string	Number of characters in the string as an integer.
# tuple	Index of highest defined element in the tuple, zero if applied to the null tuple. This is equal to the number of defined elements in the tuple in the case where there are no "holes".
abs integer	Yields the absolute value of the integer operand. Returns the value unchanged if it is zero or positive, and negates it if it is negative, so that the result is always non-negative.
abs real	Yields the absolute value of the real operand. Returns the value unchanged if it is zero or positive, and negates it if it is negative, so that the result is always non-negative.
abs string	The operand value must be a one character string or an error results. The returned result is the internal integer code for the character. Note that **abs** and **char** are inverse operators.
acos real	Yields the arccosine of the real operand which is given in radians. An error results if the operand is out of range.
arb set	The result is an arbitrary element from the set. If the set is the null set, then the result is **om.**
asin real	Yields the arcsine of the real operand which is given in radians. An error results if the operand is out of range.
atan real	Yields the arctangent of the real operand which is given in radians.
ceil real	**Ceil** yields the smallest integer x such that x >= real. For example **ceil** 3.5 and **ceil** 4.0 both yield 4.0 and **ceil** −3.5 and **ceil** −3.0 both yield −3.0.
char integer	Yields a string consisting of the character whose internal code is equal to the value of the integer operand. The range of permissible input operands and their interpretation is implementation dependent. An error results if the operand is out of range.
cos real	Yields the cosine of the real operand which is given in radians.
domain set	The set operand must be a map, i.e. it must consist entirely of pairs. The reference yields the domain set as defined by the equation: **domain** set = {a : [a,b] **in** set}
even integer	Yields TRUE if the integer operand is exactly divisible by two, and FALSE if it is not divisible by two.

expr real — Yields the exponential of the real operand (i.e. the value e**operand where e is the base of natural logarithms). An error results if the computation causes real overflow.

fix real — Returns the integer part of the real operand as an integer, dropping the fractional part. The conversion is always possible, although precision may be lost in the least significant digits for numbers of large magnitude.

float integer — Converts the integer operand to its corresponding real value. If the conversion causes overflow (which is possible in the case of very large integer operands), then an error results.

is_atom any — Yields TRUE if the operand is of type atom, and FALSE otherwise.

is_boolean any — Yields TRUE if the operand is of type boolean, and FALSE otherwise.

is_integer any — Yields TRUE if the operand is of type integer, and FALSE otherwise.

is_map any — Yields TRUE if the operand is a map (i.e. it is of type set and contains only pairs as element values), and FALSE otherwise.

is_real any — Yields TRUE if the operand is of type real, and false otherwise.

is_set any — Yields TRUE if the operand is of type set, and FALSE otherwise.

is_tuple any — Yields TRUE if the operand is of type tuple, and FALSE otherwise.

floor real — **Floor** returns the largest integer x, such that x <= real. For example, **floor** 3.5 and **floor** of 3.0 are both 3.0 and **floor** −3.5 and **floor** −4.0 are both −4.0.

log real — Yields the natural logarithm of the real operand. An error results if the operand value is zero or negative.

not boolean — Yields TRUE if the operand value is FALSE and FALSE if the operand value is TRUE.

odd integer — Yields FALSE if the integer operand is exactly divisible by two, and TRUE if it is not divisible.

pow set — Returns a set whose elements are all the subsets of the set operand, including the null set. The number of elements will be 2**n where n is the cardinality of the operand. **pow** applied to the null set yields a one element set containing the null set as its value.

random integer — Returns an integer which is pseudo-randomly distributed over the range from zero to the given operand value, including both end points. For example **random** 6 will give one of the seven integers 0,1,2,3,4,5,6.

random real — Returns a real which is pseudo-randomly distributed over the range from zero to the given operand value, including zero but not including the operand value.

random set — Returns an element from the set where the choice is governed by a pseudo-random distribution which ensures that the probability of

	picking any particular element is the same as that for any other element. Contrast this with **arb** which picks an arbitrary element, but makes no similar guarantee on the distribution (and might in fact pick the same element each time).
random tuple	Returns an element from the tuple where the choice is goverened by a pseudo-random distribution which ensures that the probability of any element of the tuple (including any "holes") being picked is the same as the probability of picking any other element.
range set	The set operand must be a map, i.e. it must consist entirely of pairs. The reference yields the range set as defined by the equation: **range** set = {b : (a,b) **in** set}
sign integer	Yields one of the integer results −1, 0 or +1 depending on whether the integer operand is negative, zero or positive.
sign real	Yields one of the integer results −1,0 or +1 depending on whether the real operand is negative, zero or positive.
sin real	Yields the sine of the real operand, which is given in radians.
sqrt real	Returns the square root of the real operand. An error results if the operand value is negative.
str integer	This reference yields the decimal string corresponding to the value of the integer, preceded by a minus sign if the value is negative (but positive values are not preceded by a plus sign).
str real	Yields the decimal string corresponding to the value of the integer using a format consistent with range and precision of the implementation. Negative values are preceded by a minus sign, but postitive values are not preceded by a plus sign.
str string	If the string operand has the form of an identifier (i.e. it starts with a letter and contains only letters, digits and the underline character), then **str** returns its operand unchanged, otherwise **str** yields the string value surrounded by quotes, and with any internal quotes doubled, i.e. it yields the string denotation corresponding to the string value.
str atom	Yields a string consisting of a number sign (#) followed by an integer value which uniquely identifies the atom value.
str boolean	Yields one of the two strings '#T' or '#F' depending on whether the operand is true or false respectively.
str om	Yields the string '*'.
str set	Yields the string consisting of a left set bracket {, followed by a blank, followed by the results of applying **str** to each element in the set, the values being separated by the sequence blank, comma, blank. Following the last value is a final blank and a right set bracket }. The null set as an operand results in the string '{ }'. If the set brackets are

	unavailable, then << and >> are used as replacements.
str tuple	Similar result to **str** set except that the elements are in order by tuple index, and the brackets are either [ ] or simple parentheses if these characters are unavailable, unless the tuple has one element, in which case tuple brackets [and] are used.
tan real	Yields the tangent of the real operand which is given in radians.
tanh real	Yields the hyperbolic tangent of the real operand. An error results if the calculation causes real overflow.
type atom	Yields the string '**atom**'.
type boolean	Yields the string '**boolean'.**
type integer	Yields the string '**integer'.**
type set	Yields the string '**set'.**
type real	Yields the string '**real'**
type string	Yields the string '**string'.**
type tuple	Yields the string '**tuple**'.

4.2.2. Binary Operators

Binary operators compute a result value from two input operands. The operator token appears between the two operands which are referred to as the left and right operands of the operator. In the following list, the allowable combinations of operand datatypes are indicated. If an operator is used with a combination of datatypes not in the list, then an error results.

integer + integer	Yields the sum of the two integer operand values. Overflow is not possible, since integers can be of arbitrary magnitude.
real + real	Yields the real sum of the two operands. An error results if the addition causes real overflow.
set + set	Yields the set which is the union of the two set operands.
string + string	Yields the string which is the concatenation of the two operand strings.
tuple + tuple	Concatenates its two tuple operands.
integer - integer	Yields the difference of the two integer operands. Overflow is not possible.
real - real	Yields the real difference of the two operands. An error results if the subtraction causes real overflow.
set - set	Yields the difference between the two set operands, i.e. the set of all elements which are contained in the first operand, but not contained in the second operand.
integer * integer	Yields the product of the two integer operands as an integer. Overflow is not possible.

real * real	Yields the read product of the two operands. An error results if the multiplication causes real overflow.
set * set	Yields the set which is the intersection of the two set operands.
string * integer	Yields a string consisting of integer number of duplications of string. For example, ’ab’*3 is the string ’ababab’. If the integer operand is zero, then the result is the null string. An error results if the integer operand is negative.
integer * string	Means the same as string * integer (i.e. the two operands can appear in either order).
tuple * integer	Yields a tuple consisting of integer number of duplications of tuple. For example, 3*[1,2] is the tuple [1,2,1,2,1,2]. If the integer operand is zero, then the result is the null tuple. An error results if the operand is negative.
integer * tuple	Means the same as tuple * integer (i.e. the two operand can appear in either order).
integer / integer	Yields the quotient of the two integer operands as a real. An error results if the divisor is zero, or if the division causes real overflow, or if either operand is outside the range of values which can be converted to real.
real / real	Yields the real quotient of the two operands as a real. An error results if the divisor is zero, or if the division causes real overflow.
integer ** integer	Yields the integer result of exponentiating the left operand by the right. Overflow is not possible. An error results if the right operand is negative, or if both operands are zero.
real ** integer	Yields the real result of exponentiating the left operand by the right operand. An error results if the exponentiation causes real overflow, or if the right operand is negative, or if both operands are zero.
real ** real	Yields the real result of exponentiating the left operand is complex, or in the case of zero to the power of zero, or if the result causes real overflow.
any#?#any	Yields the left operand if it is defined, or the right operand if the left operand is undefined **(om).**
any = any	Yields TRUE if both operands have the same type and value, or if both operands are undefined **(om)** and FALSE otherwise.
any /= any	Yields FALSE if both operands have the same type and value, or if both operands are undefined **(om)** and TRUE otherwise.
integer < integer	Yields TRUE if the left operand is less than the right operand and FALSE otherwise.
real < real	Yields TRUE if the left operand is less than the right operand and FALSE otherwise.

string < string	Yields TRUE if the left operand is lexically less than the right operand. The lexical ordering of characters is implementation dependant. Strings are compared left to right. A string is considered to be less than a longer string which matches its initial characters (e.g. 'abc' is less than 'abcf').
integer <= integer	Yields TRUE if the left operand is less than the right operand, or if the operands are equal, and FALSE otherwise.
real <= real	Yields TRUE if the left operand is less than the right operand, or if the operands are equal, and FALSE otherwise.
string <= string	Yields TRUE if the left operand is lexically less than the right operand, or if both operands are equal, and FALSE otherwise.
integer > integer	Yields TRUE if the left operand is greater than the right operand, and FALSE otherwise.
real > real	Yields TRUE if the left operand is greater than the right operand, and FALSE otherwise.
string > string	Yields true if the left operand is lexically greater than the right operand, and FALSE otherwise.
integer >= integer	Yields TRUE if the left operand is greater than the right operand, or if the operands are equal, and FALSE otherwise.
real >= real	Yields TRUE if the left operand is greater than the right operand, or if the operands are equal, and FALSE otherwise.
string >= string	Yields TRUE if the left operand is lexically greater than the right operand, or if the operands are equal, and FALSE otherwise.
boolean **and** boolean	Yields TRUE if both operands have the value TRUE and FALSE otherwise. The right operand is not evaluated (and therefore not even required to be of type boolean) in the event that the left operand is FALSE.
real **atan2** real	Yields the arc tangent of the quotient of the operands, taking into account their signs. The result is yielded in radians.
integer **div** integer	Yields the integer part of the quotient of the two operands as an integer. Overflow is not possible. An error results if the divisor is zero. The treatment of negative operands, which is not consistent with that of the **mod** operator is shown by the following table of examples:

+7 **div** +3	= +2
−7 **div** +3	= −2
+7 **div** −3	= −2
−7 **div** −3	= +2

boolean **impl** boolean	Yields FALSE if the first operand is TRUE and the second operand is FALSE, and TRUE otherwise.
string **in** string	Yields TRUE if the left operand appears as a substring of the right operand, and FALSE otherwise. For the case of a single character left operand, the effect is to test whether the character appears in the string.
any **in** set	Yields TRUE if the left operand appears as an element in the right operand, and FALSE otherwise. An error results if the left operand is undefined **(om).**
any **in** tuple	Yields TRUE if the left operand appears as an element in the right operand, and FALSE otherwise. An error results if the left operand is undefined **(om).**
set **incs** set	Yields true if the left operand includes the right operand, i.e. if every element in the right operand appears as an element in the left operand, and FALSE otherwise.
set **less** any	If the right operand value appears as an element of the left operand set value, then the result is the set with this one element removed, otherwise the result is the unchanged left operand value. An error results if the right operand is undefined **(om).**
set **lessf** any	The left operand must be a map, i.e. a set containing only pairs. The result yielded is the set of all pairs in this map operand value whose first element does not match the right operand value. If the right operand does not appear in the domain of the left operand value, then the left operand value is yielded unchanged. An error results if the right operand is undefined **(om).**
integer **max** integer	Yields the larger of the two integer operands.
real **max** real	Yields the larger of the two real operands.
integer **min** integer	Yields the smaller of the two integer operands.
real **min** real	Yields the smaller of the two real operands.
integer **mod** integer	Yields the integer remainder or modulus from dividing the first operand by the second. An error results if the divisor is zero or negative. The result is always positive, so that, for example, −7 **mod** 5 has the value +3.
string **notin** string	Yields FALSE if the left operand appears as a substring of the right operand, and TRUE otherwise. For the case of a single character left operand, the effect is to test whether the character is not included in the string.
any **notin** set	Yields FALSE if the left operand is contained as an element in the right operand, and TRUE otherwise. An error results if the left operand is undefined **(om).**
any **notin** tuple	Yields FALSE if the left operand is contained as an element in the right operand, and TRUE otherwise. An error results if the left operand is

	undefined **(om).**
integer **npow** set	Yields the set containing all subsets of the right set operand whose cardinality (number of elements) matches the integer value given as the left operand. An error results if the left operand is negative.
set **npow** integer	Means the same as integer **npow** set (i.e. the operands may appear in either order).
set **subset** set	Yields TRUE if the left operand is a subset of the right operand (i.e. if every element of the left operand appears as an element of the right operand), and FALSE otherwise.
set **with** any	Yields the value obtained by adding the value of the right operand as an element to the left operand set value. If this value is already in the set, then the first operand value is yielded unchanged. An error results if the right operand is undefined **(om).**
tuple **with** any	The result is a tuple whose length is one longer than the original left operand value as a result of concatenating the right operand value as a new last element. An error results if the right operand is undefined.

4.2.3. Compound Operators

For each binary operator, including user defined binary operators from **op** declarations, a corresponding compound operator exists whose name is formed by appending a / (slash) to the usual operator name, for example **/ is the name of the compound operator corresponding to the exponentiation operator. The slash may either immediately follow the operator name, or blanks may intervene (i.e. the operator name is actually composed of two separate tokens).

The compound operator may be used in one of two ways. It can be used as though it were a unary operator. In this case, the operand must be either a set or tuple, i.e. the expression form is one of:

```
bop / set
bop / tuple
```

where bop is the name of the base operator. The effect is to calculate a result by applying the binary operator successively to the elements of the set or tuple:

```
e1 bop e2 bop e3 bop ...
```

where e1,e2,e3 are successive elements of the set or tuple. In the case of a set these elements are in an arbitrary order. In the tuple case, the elements are in sequence and include undefined **(om)** values from "holes" in the tuple. Normal usage is that in the case of sets, it only makes sense to use an operator which is commutative (because of the arbitrary order of the elements). In the tuple case, the use of a non-commutative operator is well defined although unusual. Since most operators cause an error if given undefined operands, the use of a compound operator form with a tuple operand which contains holes will usually cause an error.

If there is only one element in the tuple or set, then this element value is returned as the result (and the operator is never applied). If the unary compound operator form is applied to a null set or tuple, the result is undefined (i.e. **om**).

The other form of use of the compound operator is binary with two operands. The right operand must be a set or tuple, as in the unary case, but the left operand may be any value:

```
any bop / set
any bop / tuple
```

The effect is to include the value of the left operand as the first operand value in the computed sequence:

```
any bop e1 bop e2 bop ...
```

where e1,e2,.. are elements from the set or tuple operand as in the unary case. If the binary form is applied to a null set or null tuple, then the left operand value is yielded as the result. The left operand thus acts as an initial value for the computation. If the operator is commutative and associative, then this value is often the identity element, for example 0 in the case of addition.

The following examples indicate some of uses of the compound operator form. Note that the set or tuple operand may be an explicit set or tuple former to obtain the effect of a direct iteration:

```
a := +/t;                          $ sum of elements in t, om if t=[ ]
a := 0+/t;                         $ same, but 0 if t=[ ]
0+/[a(i) * b(i) : i in {1..#a}]    $ dot product
min/s                              $ minimum element in a set
*/[x in t | x /= 0]                $ product of non-zero elements in t
```

If the iteration is explicitly given using a former, a tuple former should normally be used. If a set former is used then duplicate elements will only appear once:

```
0+/{x in t | x > 0}$ duplicate elements not included
```

A more detailed and precise description of the meaning of compound operator forms can be found in appendix nn where a translation in terms of equivalent SETL code is given.

4.3. Assignment Statements

Assignment statements in SETL have the general form:

```
lhs := expression;
```

The expression is formed from basic operands and operators as described in the previous sections. Lhs is the target of the assignment, and has several possibilities as described later on. The semicolon at the end terminates the statement and is considered to be part of it. All statements in SETL are terminated by a semicolon.

The basic action of an assignment statement is to compute the value represented by the right hand side expression and then to perform the assignment. The assignment action depends on the form of the left side which has several different possibilities.

Identifier

If the left hand side is a simple identifier, then the result is simply to assign the value of the right hand side as its new value, the old value being lost. This may have the effect of dynamically changing the datatype of the identifier.

String Slice

A string slice can only be used on the left side if the corresponding right hand side is a string value, otherwise an error results. The effect is to modify the string value by replacing the selected slice with the right hand side value. This may cause the length of the string to be increased or decreased as shown in the following examples:

```
a := 'ABCDEF';
a(2..4) := 'XY';      $ a = 'AXYEF'
a(4..) := '1234';     $ a = 'AXY1234'
a(2) := '()';         $ a = 'A()Y1234'
```

As indicated in the above examples, the usual defaults are permitted. An error results if the specified limits do not select a defined substring of the original string. It is possible to specify a null string by using a value for the lower bound which is exactly one greater than the upper bound. The effect is specified by the following rule:

```
a(b..c) := d    means   a := a(1..b-1) + d + a(c+1..#a)
```

Tuple Slice

A tuple slice can be used on the left side of an assignment only if the corresponding right hand side value is a tuple. The effect is similar to the use of a string slice on the left side:

```
t := [10,20,30,40];
t[2..2] := [5,6,7];     $ t = [10,5,6,7,30,40]
t[3..] := [7,8];        $ t = [10,5,7,8]
t[1..2] := [ ];         $ t = [7,8]
```

Tuple Selection

A tuple selection as a left side for an assignment allows a single specified element of a tuple to be modified:

```
t := [10,20,30,40];
t(2) := 15;             $ t = [10,15,30,40]
t(4) := om;             $ t = [10,15,30]
```

```
t(6) := 5;                $ t = [10,15,30,om,om,5]
```

As shown in the above examples, the assignment may cause the length to decrease (if the last element is assigned the value **om**), or increase (if an assignment is made to a previously non-existent element).

Tuple An enumerated tuple can be used as a left hand side of an assignment if each element of the enumeration is itself a valid left hand side. The corresponding right hand side must itself be a tuple and the effect is to perform a series of assignments using corresponding values extracted from the right hand side tuple.

```
a := [10,20];
[b,c] := a;        $ b = 10, c = 20
[x,y] := [2,3];    $ x = 2, y = 3
[c,d] := [d,c];    $ interchanges c,d
```

The last example works because the right hand side is evaluated first, forming a tuple with copies of the old values of c and d before the assignment is made.

It is permissible for the tuple on the right to be longer (extra values are ignored), or shorter (**om**'s are supplied) than the tuple on the left. It is also possible to use a minus sign as one of the components of the left side tuple, indicating that the corresponding assignment should be skipped:

```
[a,b,c] := [1,2];           $ a = 1, b = 2, c = om
[a,b] := [10,20,30];        $ a = 10, b = 20
[a,–,c] := [10,20,30];      $ a = 10, c = 30
```

A special case occurs when the right hand side is **om.** The effect if to undefine all the left side values:

```
[a,b,c] := om          $ a = b = c = om
```

Single Valued Map Reference

If a single valued map reference is used as a left side, then the effect is to remove the pair which starts with the specified domain value (if it is already present), and add the pair corresponding to the new value.

```
m := {[1,2], [2,4], [3,5]};
m(1) := 5;                 $ m = {[1,5], [2,4], [3,5]}
m(4) := 9;                 $ m = {[1,5], [2,4], [3,5], [4,9]}
```

If the right hand side is **om,** then the pair is still removed, but no pair is added:

```
m(2) := om;                $ m = {[1,5], [3,5], [4,9]}
```

An error results if the original set value is not a single valued map, i.e. if it contains any element which is not a two element tuple, or if two pairs have the same first element value.

Multi-Valued Map Reference

If a multi-valued map reference is used as a left side, then the effect is to remove all existing pairs starting with the specified domain element, and add the set of pairs specified by the right side value:

```
m := {[1,1],[1,2],[2,4]};
m{1} := {3,4};              $ m = {[1,3],[1,4],[2,4]}
```

If the right hand side is not a set, or if the left side set value is not a map, then an error results. If the right hand side is the null set, then no pairs are added:

```
m{1} := { };                $ m = {[2,4]}
```

Multiple Subscripting

In the above examples of subscript forms on the left side:

(x..y)	*String slice*
(x..y)	*Tuple slice*
(x)	*Tuple selection*
(x)	*Single valued map reference*
{x}	*Multi-valued map reference*

the subscripted quantity was a simple identifier. SETL actually permits this quantity to be itself any valid left hand side. The meaning of such a compound assignment is described by the following equivalence, where (x) stands for any of the above subscript forms:

```
lhs(x) := y;
```

means

```
t0 := lhs;
t0(x) := y;
lhs := t0;
```

Since the lhs may itself have compound form, this definition is recursive, and the third line of the translation may itself need expanding.

Although this definition seems complex, its aim is actually simple. The object of the assignment statement is to make the left side have the appropriate value after the assignment is complete, and this effect is achieved even in complex cases by the above rule.

Multi-variable Map References

The same short hand notations for dealing with maps of more than one variable which can appear in expressions are also permitted as left hand sides of assignments with similar meaning:

```
a(b,c..n) := x;      $ means a([b,c..n]) := x;
a{b,c..n} := x;      $ means a{[b,c..n]} := x;
```

4.3.1. Assigning Operatoss

For each defined binary operator, including user defined operators introduced with an **op** statement, there is a corresponding assigning operator, whose name is formed by appending the characters := to the operator name. The effect of such an operator is to perform the usual computation and then assign the result back to the left operand, which must therefore have the form of a valid assignment left hand side. Most often these assigning operators are used directly in statement forms as in the following examples:

```
a max:= 5;     $ set a to maximum of current value and 5
b with:= 6;    $ adds the element 6 to the set b
i +:= 1;       $ increment integer value of i by one
```

It is also possible to use assigning operators within an expression, in which case the value yielded is the computed result and the assignment occurs as a side effect:

```
a(i+:=1) := 5;    $ increment i by 1, assign i'th element
```

The assignment operator itself is a special case of an assigning operator and thus may also be used within an expression, the value being the assigned result:

```
a:=b:=1;       $ multiple assignment, a = b = 1
a:=3+c:=5;     $ c = 5, a = 8
```

The following special assigning operators modify both their operands which must therefore both have the form of valid assignment left hand sides. It is not possible to append := to any of these operator names, nor is it possible to form compound operator forms by appending a slash.

lhs **from** set	an arbitrary element (in the same sense as the **arb** operator) is selected from the right hand set operand if it is non-null. This value is assigned as the new value of lhs. The right operand is assigned a new set value consisting of the old set value less the selected element. If the right operand is the null set, then the lhs is set to undefined **(om)** and the right operand value is unchanged.
lhs **fromb** tuple	If the right operand is not the null tuple, then the first element of the tuple (i.e. the element indexed by the index value 1) is assigned as the value of lhs, and the right operand is assigned a new tuple value consisting of the remaining elements of the tuple. If the right operand is the null tuple, then lhs is set to undefined **(om)** and the right operand value is unchanged.
lhs **fromb** string	If the right operand is not the null string, then the first character of the string is assigned as the value of lhs, and the right operand is is assigned a new string value consisting of the remaining characters of the string. If the right operand is the null string, then lhs is set to undefined **(om)** and the right operand value is unchanged.
lhs **frome** tuple	If the right operand is not the null tuple, then the last element of the tuple is assigned as the value of lhs, and the right operand is assigned a new tuple value obtained by removing the last element (i.e. setting it to undefined). If the right operand is the null tuple, then lhs is set to undefined **(om)** and the right operand value is unchanged.
lhs **frome** string	If the right operand is not the null string, then the last character of the string is assigned as the value of lhs, and the right operand is assigned

a new string value consisting of the remaining characters. If the right operand is the null string, then lhs is set to undefined **(om)** and the right operand value is unchanged.

4.3.2. Quantified Tests

Quantified tests are special expression forms yielding a boolean result (TRUE or FALSE) and which therefore can be used in contexts requiring a test. Quantified tests involve an implied loop and appear in three forms:

```
exists iterator | test
notexists iterator | test
forall iterator | test
```

The syntax and interpretation of the iterator is further described in the section on iterator forms which appears in the next chapter. As indicated, the iterator must contain a such that test preceded by the | character (or the equivalent **st** keyword).

In each case, the iterator causes an implied loop to be executed.

In the case of the **exists** test, the result is TRUE if the test succeeds (yields TRUE) on any of the loops. As soon as the test succeeds, the iteration is abandoned (leaving any iteration variables set to the values for which the test succeeded) and the result of the expression is TRUE. If the test fails for all loops, or if the loop is not executed at all, then the result is FALSE and the iteration variables, if any, are set to undefined. The **notexists** form executes the same test loop, but yields TRUE if the test never succeeds and FALSE if the test does succeed one some iteration.

```
if exists i in [1..#t] | t(i) /= 0 then
  ...                         $ with i = 1st non-zero element in t
else
  ...                         $ if no non-zero elements, i = om
end if;

if notexists [a,b] in m | a=b then
  ...                         $ if no such element, a=b= om
else
  ...                         $ a,b set to element with a=b
end if;
```

The **forall** test also sets up a loop. For each iteration of the loop, the specified test is evaluated. If this test yields FALSE on any iteration of the loop, then evaluation of the loop is immediately terminated, the result of the **forall** test is FALSE, and the loop variables, if any, are left with the values causing the test to fail. If the loop terminates with the test condition evaluating to TRUE for all iterations, then the result of the **forall** test is TRUE. A special case of this last rule arises when there are no iterations (e.g. a test on a null set). Such a test always returns a result of TRUE.

```
if forall i in [1..#t-1] | t(i) < t(i+1) then
  ...                                   $ tuple elements are sorted, i= om
else
  ...                                   $ i is index of out of order pair
end if;

if forall x in s, y in s | (x .op y) in s then
  ...                                   $ s is closed under operator .op
else
  ...                                   $ x,y set to counter example
end if;
```

It is possible to use these quantified test forms as parts of larger expressions, but parenthesization is required except when a quantified test is used as the test in another quantified test:

```
exists a in x | c(a) and d(a)
exists a in x | (c(a) and d(a))                 $ means the same
(exists a in x | c(a)) and d(a) $ to get the other meaning

        if exists a in c |
            exists b in x |
                forall y in z :
                    exists r in s | a = r then ...
```

4.3.3. Operator Precedence Rules

The table in this section shows the precedence rules which determine the order in which the operators in an expression are evaluated. If two operators share a common operand, then the one with the higher precedence is evaluated first. If both operators have the same precedence, then the left hand one is evaluated first (i.e. operators of a given precedence level are evaluated in a left associative manner.)

Parentheses may be used freely to emphasize or alter the order of operations as determined by this table.

Precedence	*Operators*
11	:= (on left side) assigning operators (on left side) **from** (both sides)
10	All unary operators except **not** and the **is**_xx operators.
9	**
8	* / **mod div**
7	+ −
6	User defined binary operators
5	= /= < <= > >= **in notin subset incs**
4	**not** and the **is**_xx operators
3	**and**
2	**or**
1	**impl**
0	:= (on right side) assigning operators (on right side)

The following examples of equivalent expressions with and without parentheses illustrate the operation of these rules:

a + b + c * d
(a + b) + (c * d)

a + b +:= c **div** d
a + (b +:= (c **div** d))

a + **ceil** b := c
a + (**ceil** (b := c))

Compound operators used in the unary form (with no left operand) have the same precedence as other unary operators. Compound operators used in the binary form have the same precedence as the operator from which they are constructed.

The precedence rules also exclude the use of quantified tests as operands of any operator, unless they are enclosed in parentheses. The only case in which a quantified test can appear as part of a larger expression is when one test appears as the iterator test of another quantified test.

4.3.4. Side Effects

The fact that assigning operators can be used within expressions gives rise to questions about possible orders of side effects. For example, what is the value of the expression:

```
(i +:= 3) + i + (i := 5)
```

and what is the value of i after computing this expression?

SETL deals with this question by specifying that the order of operations within a single expression is unpredictable. This means that any expression having side effects whose result depends on the order in which the various parts of the expression are evaluated gives an undefined result.

In general, the use of multiple assignments to the same variable within a single expression may give rise to such an undefined result and should be avoided. It is also risky to assign a value to a variable in one part of an expression and reference this same variable in another part of the expression which may be evaluated before or after the assignment.

Another case of possible side effects arises in connection with the left operand of an assigning operator which has a side effect itself, typified by the following question:

```
a(x)+:=1 means a(x):=a(x)+1
```

How many times does a(j+:=1)+:=1 increment j?

The rule followed here is that the result is undefined if the answer to the question matters, that is if the result of the assignment, or the resulting value of any identifiers assigned as a side effect, depends on whether there are one or two evaluations, then the result is undefined. Thus the assignment a(j+:=1)+:=1 is invalid.

The purpose of making all these nasty cases undefined is to give the compiler freedom to choose the easiest or most efficient translation without being hampered by worrying about peculiar cases which do not occur in practice. The programmer's job in this respect is to avoid "tricky" use of side effects and make sure that these cases do not appear at all!

CHAPTER 5

CONTROL STATEMENTS

This chapter describes the statements in SETL which are used to control the flow of processing.

5.1. Conditional Statements & Expressions

The conditional statements in SETL allow the flow of control to be determined on the basis of specified tests. Conditional expressions allow selection of a calculation in an expression depending on the outcome of a test.

5.1.1. If Statement

A basic control structure in SETL is the **if** conditional statement, which allows selection among sequences of program statements to be executed depending on the outcome of a test. The simplest form is a two way **if** statement:

```
if test then
    block1
else
    block2
end if;
```

The test after the **if** is evaluated. If it succeeds, then the series of statements (zero or more) which comprise block1 are executed. If the test fails, then the series of statements in block2 are executed. The following are some examples of this two way **if** structure:

```
if a > 3  or  status = 'error' then
    print('BAD A VALUE OR OTHER ERROR');
    a := 3;
    status := 'ok';
else
    x(a) := 2;
    print(y(a));
end if;
```

```
if marked('x') then stop; else goto x; end if;

if (a > 3  or  b < 2) and not c > 3 then
    d := a + b + c;
else
    a := 2;
    b := 0;
end if;
```

The indentation used in the above examples (except for the last one which can fit all on one line) is not required by the SETL language, but it represents a recommended style which improves readability of programs.

The **else** keyword and its following statements in block2 may be omitted, in which case no action occurs if the test fails:

```
if d = 0 then
    print('zero diagonal element');
    stop;
end if;
```

The **end if** terminator may be written in two other styles. It can be shortened to **end;** which is useful in the case of short **if** statements where no confusion can arise. Alternatively it can be expanded by copying one or more tokens from the test in the corresponding **if** as shown in the following examples:

```
if a > 3 then print(a); end if;
if a > 3 then print(a); end if a;
if a > 3 then print(a); end if a > 3;
```

This expanded form is particularly useful for very long if statements, since it helps the reader to match up **end**'s with their corresponding **if** tests. It also helps the compiler perform this match up if errors have been made in the program.

Since an **if** statement is considered to be a single statement itself, ending with a semicolon as usual, it may appear as one of the statements in the **then** or **else** block, allowing nesting of conditional tests to an arbitrary level. A special case of such nesting occurs when several conditions are tested in sequence as in the following example.

```
if day = 'sun' then
    d := 1;
else
    if day = 'mon' then
        d := 2;
    else
        if day = 'tue' then
            d := 3;
```

```
            else
                if day = 'wed' then
                    d := 4;
                else
                    if day = 'thu' then
                        d := 5;
                    else
                        if day = 'fri' then
                            d := 6;
                        else
                            d := 7;
                        end;
                    end;
                end;
            end;
        end;
    end;
```

Although this achieves the intended effect, it illustrates the awkwardness of making sure that all the **end** statements match up. The following alternative approach is easier to write, but less efficient since all the tests are made in every case.

```
if day = 'sun' then d := 1; end;
if day = 'mon' then d := 2; end;
if day = 'tue' then d := 3; end;
if day = 'wed' then d := 4; end;
if day = 'thu' then d := 5; end;
if day = 'fri' then d := 6; end;
if day = 'sat' then d := 7; end;
```

Furthermore, there are cases, unlike this example, where it is not merely inefficient, but definitely wrong, to perform a test when an earlier test has succeeded.

A special keyword **elseif** allows the sequence to be written in an efficient manner without having to count **end** statements. Whenever **else** and **if** keywords appear immediately adjacent to one other, they may be replaced by the single keyword **elseif,** and the corresponding **end** terminator is omitted. Thus we can write:

```
if day = 'sun' then d := 1;
elseif day = 'mon' then d := 2;
elseif day = 'tue' then d := 3;
elseif day = 'wed' then d := 4;
elseif day = 'thu' then d := 5;
elseif day = 'fri' then d := 6;
elseif d := 7;
end;
```

In this form, the tests are in sequence and as soon as one test is TRUE and its corresponding block is executed, the remainder of the tests are abandoned.

5.1.2. If Expression

A special form of expression called a conditional expression or **if** expression is similar in form to an **if** statement:

if test **then** expr1 **else** expr2 **end**

This may be used anywhere that a expression is permitted. The effect is to evaluate either expr1 or expr2 depending on the result of the test. The result of the entire **if** expression is then the value of the single expression evaluated. The terminator must be **end** as shown (not **end if** or **end if** with extra tokens). Some examples of conditional expressions are:

```
print(if a > 5 then b else f(a) end);
maxab := if a > b then a else b end;
```

5.1.3. Case Statement

The **case** statement is a generalization of the **if** concept which allows one of several sequences of statements to be selected depending on a set of associated tests. The general form is:

```
case of
(test1): block1
(test2): block2
(test3): block3
(testn): blockn
else blocke
end case;                    $ or simply end;
```

Each of block1,block2,.. and blocke is a sequence of one or more statements. Each of the expressions test1,test2,.. is evaluated and must yield a value of TRUE or FALSE. If none of the expressions yields TRUE, then the **else** block, blocke, is executed. If one or more of the tests yields TRUE, then one of the associated blocks of statements is executed. If more than one test yields TRUE, the choice of which block to execute is made in an arbitrary manner. The **case** statement thus differs from a similar sequence of **if** and **elseif** tests where the tests are made in sequence.

It is possible to attach more than one test to a given branch of the case by:

```
(test1,test2,.. testn): blockn
```

In this case, blockn is a candidate for being executed if any one of the tests test1,test2,.. yields TRUE.

The **else** clause, together with its associated block of statements can be omitted, in which case, the **case** statement has no effect if none of the tests evaluates to TRUE.

If there is at least one test which evaluates to TRUE, then the other tests may or may not be evaluated. Effectively the separate tests are evaluated in some arbitrary order, or even at the same time in an unspecified sequence, and the program can make no assumptions about the order or about which tests are actually evaluated.

We can now write the days of the week example as a **case** statement:

```
case of
(day='sun'): d := 1;
(day='mon'): d := 2;
(day='tue'): d := 3;
(day='wed'): d := 4;
(day='thu'): d :(eq 5;
(day='fri'): d := 6
(day='sat'): d := 7;
end case;
```

In this particular case, at most one of the **case** branches could be true, so there is no difference in effect between this **case** statement and the sequence of **if** and **elseif** tests given in the previous section. The following example shows a case in which there is a difference:

```
if k>10 then
    print('k>10');
elseif k<5 then
    print('k<5');
elseif k<10 then
    print('k<10');
else
    print('k=10');
end if;

case of
(k>10):
    print('k>10');
(k<5):
    print('k<5');
(k<10):
    print('k<10');
else
    print('k=10');
end case;
```

If these two sequences are executed with k having the value 4, then the top example using **if** and **elseif** will always print k<5 since this test is always made before the k<10 test. In the CASE statement, either k<5 or k<10 might be printed since both conditions are satisfied and the order in which these two tests are made is not specified. A general rule is to use **if** and **elseif** only when the order is important. This serves to warn the reader of the program that the order of test evaluations is important.

One particular application of the undefined order in which the tests are made is illustrated by the following:

```
case of
(true):
(true):
end case;
```

This construction can be used if two possible sequences of statements can be executed and it does not matter which is executed, or in any case the programmer does not wish to indicate a choice between the two possibilities. The implication is that the program will work no matter which of the two branches is executed. This gives a control flow analog to the use of the **arb** operator in expressions and allows the programmer to avoid specifying choices which are not important to the algorithm. A similar situation might arise in a sorting program where the equal comparison condition can be treated as either less than or greater than:

```
case of
(a(i) <= a(i+1)):
(a(i) >= a(i+1)):
end case;
```

It is quite common for the tests in a **case** statement to have the form of testing some particular variable or expression value for equality with a series of constants. The days of the week example is in this form. A special form of the **case** statement is available to simplify the writing of such cases:

```
case expr of
(constant1): block1
(constant2): block2
(constant3): block3
(constantn): blockn
else blocke
end case;                $ or end; or end case <tokens>;
```

The expression in the header expr is evaluated (once) to give a test value. A block is executed if its associated constanti expression is equal to the test value. As in the full form of the **case** statement, if more than one value matches, then an arbitrary choice is made between the matching alternatives. The **else** block is executed if no values match and can be omitted if no action is required in the case where no values match. As in the full form, multiple tests can be attached to one branch by:

```
(constant1,constant2,...constantn): block
```

in which case the block is a candidate for execution if any of the attached values match.

The actual rules for what can appear as constants here are the same as for **const** declarations, i.e. denotations, previous declared constant values, or sets or tuples of constant values.

We can now write the days of the week example in a more compact form using this version of the **case** statement:

```
case day of
('sun'): d := 1;
('mon'): d := 2;
('tue'): d := 3;
('wed'): d := 4;
('thu'): d := 5;
('fri'): d := 6;
('sat'): d := 7;
end case day;
```

Note that the expressions attached to the branches in this form are general expressions and are not required to be constants, although the use of constants in this context is common. As in the full form of the case, no assumptions can be made about the order of evaluation of these expressions, or even about whether or not they will be evaluated if there is a matching alternative.

5.1.4. Case Expression

Just as there is an **if** expression corresponding to an **if** statement, there is a **case** expression which corresponds to a **case** statement. The form is:

```
case expression of
(value1): expression1,
(value2): expression2,
(value3): expression3,
(valuen): expressionn
else expressione
end
```

The value of the **case** expression is the value of the selected expression, or the value of the **else** expression if no expressions match. If the **else** and its following expression are omitted, and there is no matching entry, then the result of the **case** expression is **om.** As in the case statement, a single expression branch may be labeled with more than one constant. Note that there is no comma following the last branch of the **case** (whether or not an **else** is present).

We will complete this section by rewriting the days of the week one final time using a **case** expression:

```
d := case day of
                ('sun'):     1,
```

```
                ('mon'):   2,
                ('tue'):   3,
                ('wed'):   4,
                ('thu'):   5,
                ('fri'):   6,
                ('sat'):   7
    end;
```

5.2. Loop Statement

The loop statement in SETL allows a series of one or more statements to be executed repeatedly. There are two basic forms. The first form uses an iterator:

```
loop for iterator do
  ...                   $ loop body
  ...                   $ zero or more statements
end loop;               $ or end; or end loop tokens;
```

In addition, parentheses can be used to replace the **oop** and **do**:

```
(for iterator)
  ...
  ...
end;                    $ or end tokens;
```

In this case, the token sequence after **end** copies tokens starting immediately after the left parenthesis which opens the loop.

The iterator, whose possible forms are discussed in the following section, controls the number of times the loop is executed and also may assign values to one or more iteration variables.

The other form of loop is written:

```
loop
  init blocki     $ loop initialization statements
  doing blockd    $ step statements at start of loop
  while testw     $ termination test at start of loop
  step blocks     $ step statements at end of loop
  until; testu    $ termination test at end of loop
  term blockt     $ loop termination statements
do
  blockb          $ body of loop
```

```
end loop        $ or end; or end loop tokens;
```

As with the iterator form of loops, the keywords **loop** and **do** can be replaced by parentheses:

```
(iniut b doing b while t step b until t term b)
end;                $ or end tokens;
```

The easiest way of describing the precise effect of this loop form is to write out the sequence of statements which are executed when this form is used to construct a loop:

```
        blocki              $ init block
head:
        blockd              $ doing block

        if not (testw)      $ while test
        then goto term;
        end if;

        blockb              $ body of loop

step:
        blocks              $ step block

        if (testu)          $ until test
        then goto term;
        end if;

        gotohead;

term:
        blockt              $ term block
```

If a **quit** statement is executed within the loop (i.e within the **doing** or **step** blocks or in the loop body), then it is equivalent to **goto** term in the above translation. A **continue** statement is similary equivalent to **goto** step.

If any of the **init, doing, step** or **term** clauses are omitted from the iterator, then the corresponding block is simply removed from the loop. Similarly with the **while** and **until** tests, if either of these is omitted then the corresponding test is omitted from the loop. Some common forms of loops using simple cases of the iterator construction are as follows:

```
loop while test do    $ while loop, test at start
loop until test do    $ until loop, test at end
```

loop do $ indefinite loop

5.2.1. Quit & Continue Statements

Within a loop, two special statements are available. The **quit** statement causes immediate termination of execution of the loop, and is written:

quit;
quit loop;
quit tokens;
quit loop tokens;

The last forms are useful in the case where loops are nested. By copying tokens from the corresponding iterator (including the token **loop** only if the loop was written in the **loop-do** form), it is possible to specify which loop is to be terminated.

The **continue** statement causes the current iteration of the loop to be abandoned, and the execution of the loop continues with the next iteration (if there is one left to go). The form is:

continue;
continue loop;
continue tokens;
continue loop tokens;

As in the case of the **quit** statement, the forms which copy tokens from the corresponding loop iterator can be used to specify which loop is to be continued in the case of nested loops.

Normally **quit** and **continue** statements appear within the body of the loop, but they can also occur in the **doing** and **step** forms of the alternate loop form. Note that they cannot occur in the **init** or **term** blocks (unless referring to a surrounding loop).

The use of the **quit** or **continue** statements other than statically inside a loop is erroneous. In particular, it is not possible to execute the **quit** or **continue** from a procedure or refinement called within the loop body.

5.2.2. Iterator Forms

Iterators are used to control loops, in set and tuple formers, and in quantified tests **(exists, forall).** The form of an iterator determines the number of times the associated loop is executed and also assigns values to one of more identifiers called iteration variables.

The following section describes the basic iterator forms. More precise definitions of these iterator forms can be found in Appendix nn, where translations into equivalent loop forms with explicit step statements and tests are given.

lhs **in** set — The iteration variable lhs is set to successive values from the set, chosen in arbitrary order. The number of iterations is equal to the number of elements in the set. One particular form which is useful in iterating through a map uses a lhs of the form [a,b] which causes a and b to be

	set to successive domain and range values.
lhs **in** string	The iteration variable is set to successive characters of the string value starting with the first and ending with the last. The number of iterations is equal to the length of the string. No iterations are performed if the string is null.
lhs **in** tuple	The iteration variable lhs is set to successive elements of the tuple in order by increasing index. The number of iterations is equal to the index of the highest defined element. If there are no holes (embedded undefined values), then this is equal to the number of defined elements. If there are holes, then the iteration variable will be set to **om** for the undefined index positions.
lhs1 = map(lhs2)	This form is used to iterate through the elements of a map, which must be single valued. The iteration variables lhs1 and lhs2 are set to successive domain and range values respectively, the order in which pairs are taken being arbitrary. The number of iterations is equal to the number of elements in the map.
lhs1 = map{lhs2}	This form of map iterator can be used with multi-valued maps. On successive iterations, lhs1 is set to the values in the domain of the map and lhs is set to the corresponding image set. The number of iterations is thus equal to the cardinality of the domain.
lhs1 = string(lhs2)	The iteration variable lhs2 is set to successive integer values 1,2... up to the length of the string. The iteration variable lhs1 is set to the corresponding character from the string. The number of iterations is thus equal to the length of the string.
lhs1 = tuple(lhs2)	The iteration variable lhs2 is set to successive integer values 1,2,.. up to the highest index for which the tuple is defined (which is therefore the number of iterations). The iteration variable lhs1 is set to the corresponding tuple value.

In these iterator forms, the iteration variables can be any valid left hand side. For example [a,b] can be used as an iteration variable if all the iteration values are pairs, and the effect is to set a and b to successive values of the elements of the pairs. On normal completion of a loop (but not on premature termination from a **quit,** satisfied **exists** etc.) the iteration variables are all set to undefined **(om),** so the final values are not available outside the loop.

The usual shorthand notation can be used in iterator contexts for iterating through multi-argument maps:

lhs = map(a,b..n)	means	lhs = map([a,b..n])
lhs = map{a,b..n}	means	lhs = map{[a,b..n]}

It is permissible to modify the values of any iteration variables or other identifiers mentioned in the iterator, although such usage is not recommended. Such modifications do not affect the values yielded by the iteration. For example, the following iteration:

```
a := [1,2,3,4,5,6,7,8,9];
(for i in a) a:=a+a; i+:=1; end;
```

still iterates nine times with i being set to the successive values 1 through 9.

Any context requiring an iterator (loops, set formers, tuple formers, quantified tests) can use any of the iterator forms described. More complicated iterator forms can be built up from these basic iterator forms given:

Compound iterators are formed as a sequence of basic iterators separated by commas. The effect is that of a series of nested loops, in which the last iterator is stepped most rapidly and the first iterator least rapidly. Note that a **quit** statement will leave all the nested loops in this case and a **continue** will continue the inner loop (in this respect the effect is slightly different from actually writing nested loops).

Any iterator consisting of one or more basic iterator forms can be followed by a such that clause:

```
iterator | test
```

The effect is that the iterator is stepped as usual, but on each iteration test is evaluated. If the result is FALSE, then the iteration is skipped, i.e. the iterator is immediately moved to the next step. If the test is TRUE, then the iteration is completed as usual. It is often the case, but is not required, that the test will perform tests on the values of iteration variables.

The following are examples of loops constructed from iterator forms:

```
(for i in [1..#a], j in [i..#a])
    c := a(i)*a(j);
end;
(for x in y, a in b | x = a)
end;

(for x = c(i) | x = ',')
end;
```

5.2.3. Set & Tuple Formers

The effect of set and tuple formers can be described exactly by writing equivalent loop forms. First the set former:

```
{expression : iterator}
```

is equivalent to the following **expr** block:

```
expr
    t1 := { };
    loop for iterator do
        t1 with:(eq expression;
```

```
        end;
        yields t1;
    end
```

The tuple former:

    [expression : iterator]

differs only in that a tuple is formed, and is equivalent to the following **expr** block:

```
    expr
        t1 := [ ];
        loop for iterator do
            t1 with:= expression;
            end;
            yield t1;
    end
```

As implied by the translations, the result is the null set or null tuple if the iterator does not cause any iterations of the loop.

Multiple iterators are permitted, as in the following example:

    {x+y : x **in** seta, y **in** f(x) | x/=y}

If the iterator is a single (non-compound) iterator with a such that test, then the following abbreviated forms are available:

{a **in** expr | t}	$ for {a : a **in** expr | t}
[a **in** ... | t]	$ for [a : a **in** expr | t]

5.3. Goto Statement

Any statement in SETL can be labeled:

    label: statement;

As shown, a single statement can have only one label. The label names are local identifier names which are not explicitly declared and which are not used for any other purpose.

The **goto** statement, which has the form:

    goto label;

has the effect of transferring control to the labeled statement. **Goto** statements are forbidden to make any of the following types of transfer:

>From outside the body of a loop into the body
Into the **then** block of an **if** from outside
Into the **else** block of an **if** from outside
Into the **init, doing, step, until** blocks from outside
Into the blocks in a **case** or **if** statement from outside
>From one block in a **case** or **if** statement into another
>From outside an **expr** block into its body

Since all labels are local, it is also impossible to transfer from one procedure to another, or from a procedure body to the main program.

5.4. Stop Statement

The **stop** statement, written simply as:

stop;

may appear anywhere in a SETL program and causes immediate termination of program execution. The two ways of terminating a SETL program normally are by execution of a **stop** statement, or by executing "off the end" of the main program block.

5.5. Pass Statement

The **pass** statement, written as:

pass;

has no effect when it is executed. It is used for placing labels at the end of a block, for branches of a **case** statement where no action is performed, for a **loop** body when the actual processing is completed in the loop header, and in any other situation where a dummy statement is required.

5.6. Assert Statement

The **assert** statement, written as:

assert expression;

signals an error if the value of the expression is not TRUE.

CHAPTER 6

THE DATA STRUCTURE SPECIFICATION SUBLANGUAGE

Unlike most of the sections of this book so far, the title of this chapter probably seems mysterious. This is not surprising since it describes a feature of SETL which is unique and therefore unfamiliar from the use of other programming languages.

The general idea is the following. SETL allows programs to be written without much concern for the detailed data structures required for efficient execution. For example in the curriculum planning problem in chapter 2 (the topological sort), the basic data structure was a graph which showed the prerequisite structure. In most other programming languages, a program using such a graph must decide exactly how to represent the graph in the memory of the computer. This is a rather complicated matter, and there are several different ways of representing graphs (linked lists, adjacency tables, bit matrices etc.) Furthermore, once a decision is made on the exact manner in which the data structure should be represented, then the details of this representation must be kept in mind throughout the program and the text of the program reflects this choice throughout.

In SETL, such detailed decisions do not have to be made by the programmer. In this particular example, the graph was simply a set of pairs corresponding to the edges of the graph as read in. The programmer did not need to have any concern with the exact manner in which the graph would be represented. This is highly convenient from the programmers point of view, since it obviously reduces the amount of work in writing the program. However, there is a penalty paid for this approach in terms of reduced execution efficiency. The choice of data structure representations is a delicate task. The choice which leads to the most efficient program depends on the exact operations which are performed and the frequency with which the various operations are performed. Since the SETL programmer omits this choice, it is left up to the compiler. The SETL compiler is not capable of making the best possible choice since it does not have enough information, so it chooses rather general representations which have the property that they are not disastrously bad, no matter what operations are performed. On the other hand these general structures are often not perfect for the particular case at hand and the result is longer than necessary running time.

In some cases, this inefficiency may not be a problem. If a program runs in one second, it may not matter that it could have run in one tenth of a second if better data structure decisions had been made. If all the programs you write are in this category, then you can abandon reading the rest of this chapter. On the other hand, if you have programs which run for a long time, then the subject matter of this chapter will be of interest, particularly if you are paying for your own computer time!

Suppose that we have a SETL program which is running perfectly, but slowly, and the reason for the slowness can be traced t failure to use efficient data structure representations. One approach would be to reprogram in some other lower level language in which the data structure representation would be under direct control of the programmer. This would have two obvious disadvantages. First the sheer effort of dealing with another language and recoding the entire program would be a substantial effort. Second, this approach requires that once the data structure representation is chosen, it must be kept in mind throughout the program and every operation on the structure must be thought out in terms of this representation.

The data structure representation sublanguage of SETL provides an alternate approach. Instead of rewriting the entire program, statements are added which direct the compiler to use certain specific (and hopefully appropriate) data structures. Given this information, SETL can generate a more efficient program and the compiler assumes the burden of going though the program making the necessary modifications. Assuming that the programmer did indeed specify an appropriate choice, the desired improvement is thus obtained with a minimum amount of error prone clerical work on the part of the programmer and with no change to the text of the working program. The general design of the data structure specification language is that statements specifying representations cannot change the meaning of the program, the worst that can happen is that the program actually runs slower if an inappropriate choice is made.

The concept is thus reasonably straightforward, but the details of the representation language are technical and the rest of this chapter explains them. Again you are invited to skip the rest of this chapter if you can live with the efficiency you are getting now using "pure" SETL. If on the other hand, your monthly computer bills are causing headaches, then read on!

{to be supplied later}

CHAPTER 7

SEMANTIC DEFINITIONS

This chapter contains a more formal definition of most of the semantics of the SETL language. As far as possible the definitions are written in SETL. In those few cases where it is not possible to use SETL, a construction called a pseudo-comment is used. A pseudo-comment is an english language description enclosed in the brackets /* and */. It may appear in place of a statement or expression and is conceptually replaced by code which carries out the intent of the description.

One pseudo-comment which appears frequently is /*error*/ which implies that an error is caused. The exact effect of such an error depends on the implementation, in most cases, execution is terminated with an error message. However, some implementations may have the capability of continuing execution after such an error has occured.

7.1. Type Test Operators

Since the type test operators are used throughout the definitions of the remaining operators, they are defined first.

```
    op type (a);
        if a = om then /*error*/;
        else
            return
/*      one of the following strings indicating the type of the operand a:
                    ´atom' ´boolean' ´integer' ´real' ´set' ´string' ´tuple'     */ ;
        end if;
    end op type;

        op is_boolean (a);              $ test argument is type Boolean
           return type a = 'boolean';
        end op is_boolean;
```

```
op is_ integer(a);              $ test argument is type integer
   return type a = 'integer';
end op is_ integer;

op is_real(a);                  $ test argument is type real
   return type a = 'real';
end op is_real;

op is_string(a);                $ test argument is type string
   return type a = 'string';
end op is_string;

op is_atom(a);                  $ test argument is type atom
   return type a = 'atom';
end op is_atom;

op is_tuple(a);                 $ test argument is type tuple
   return type a = 'tuple';
end op is_tuple;

op is_set(a);                   $ test argument is type set
   return type a = 'set';
end op is_set;

op is_max(a);                   $ test argument has form of map
   if is_set a then
return forall x in a |
     is_tuple x and #x = 2;
   else
    return false;
   end if;
end op is_map;
```

7.2. Operator Definitions

This section contains SETL definitions of the SETL operators. The definitions are written in standard SETL with three minor extensions as follows:

The names of the operators which appear in these OP definitions are the actual operator names and do not meet the normal requirements for declared operator names.

In some cases **(from, fromb, frome),** the arguments are specified as **wr** (write) or **rw** (read/write).

In a few cases (e.g. real multiplication), the semantics of the operator cannot be specified in SETL. In such cases, a pseudo-comment is used to specify the intended result.

```
op +(a);
 case of
 (is_integer a):                          $ integer affirmation
    return a;
 (is_real a):                             $ real affirmation
    return a;
 else /*error*/;
 end case;
end op +;

op +(a,b);
 case of
 (is_integer a and is_integer b):         $ integer addition
    return a - -b;
 (is_real a and is_real b):               $ real addition
    return a - -b;
 (is_string a and is_string b):           $ string concatenation
    return /* concatenation of a and b */;
 (is_tuple a and is_tuple b):             $ tuple concatenation
    x := a;
    (for i in [1..#b])
      x(#a + i) := b(i);
    end for i;
    return x;
 (is_set a and is_set b):                 $ set union
    return (a with/ {x : x in b});
 else /*error*/;
 end case;
end op +;

op -(a);
 case of
 (is_integer a):                          $ integer negation
    return 0 - a;
 (is_real a):                             $ real negation
    return 0.0 - a;
 else /*error*/;
 end case;
end op -;

op -(a,b);
 case of
 (is_integer a and is_integer b):         $ integer subtraction
```

```
      return /* integer difference a-b */;
    (is_real a and is_real b):                  $ real subtraction
      return /* real difference a-b */;
    (is_set a and is_set b):                    $ set difference
      return (a less/ x : x in b);
    else /*error*/;
    end case;
  end op -;

  op *(a,b);
    case of
    (is_integer a and is_integer b):            $ integer multiplication
    x := (0 +/ a : 1..abs b);
      return if b>=0 then x else -x end;
    (is_real a and is_real b):                  $ real multiplication
      return /* real product a*b */;
    (is_set a and is_set b):                    $ set intersection
      return {x : x in a | x in b};
    (is_string a and is_integer b):             $ string duplication
      if b < 0 then /*error*/; end if;
      return '' +/ [a : i in {1..b}];
    (is_integer a and is_string b):             $ string duplication
      if a < 0 then /*error*/; end if;
      return '' +/ [b : i in {1..a}];
    (is_tuple a and is_integer b):              $ tuple duplication
      if b < 0 then /*error*/; end if;
      x := [ ];
    (for i in [1..#b])
        j := #x;
        (for k in [1..#a])
          x(j + k) := a(k);
        end for k;
      end for i;
      return x;
    (is_integer a and is_tuple b):              $ tuple duplication
      return b * a;
    else /*error*/;
    end case;
  end op *;

  op /(a,b);
    case of
    (is_integer a and is_integer b):            $ integer division, real quotient
      return float a / float b;
    (is_real a and is_real b):                  $ real division
```

```
      return /* quotient a/b */;
   else /*error*/;
 end op /;

 op **(a,b);
   case of
   (is_integer a and is_integer b):          $ integer exponentiation
      if b < 0 then /*error*/; end if;
      return (1 */ a : 1..b);
   (is_real a and is_integer b):             $ real to power of integer
      if b < 0 then /*error*/; end if;
      return (1.0 */ a : 1..b);
   else /*error*/;
   end case;
 end op **;

 op ?(a,b);                                  $ undefined interrogation
   case of
   (a /= om):
      return a;
   (a = om):
      return b;
   end case;
 end op ?;

 op #(a);
   case of
   (is_string a):                            $ length of string
      x := 0;
      (for y in a) x +:= 1; end;
      return x;
   (is_set a):                               $ cardinality of set
      x := 0;
      (for y in a) x +:= 1; end;
      return x;
   (is_tuple a):                             $ length of tuple
      x := 0;
      (for y in a) x +:= 1; end;
      return x;
   else /*error*/;
   end case;
 end op #;

 op =(a,b);                                  $ equality test
   case a of
```

```
  (b):
     return TRUE;
  else
     return FALSE;
  end case;
end op =;

op /=(a,b);
  return if a = b then FALSE else TRUE end;
end op /=;

op <(a,b);
  case of
  (is_integer a and is_integer b):            $ integer less than
      if /* a less then b */ then
          return TRUE;
      else
          return FALSE;
      end if;
  (is_real a and is_real b):                  $ real less than
      if /* a less than b */ then
          return TRUE;
      else
          return FALSE;
      end if;
  (is_string a and is_string b):              $ string less than
      if a = '' then
          return b /= '';
      elseif b = '' then
          return FALSE;
        else
        case of
        (a(1) = b(1)):
          return a(2..) < b(2..);
        (/* a(1) precedes b(1) in order */):
          return TRUE;
        else
          return FALSE;
        end case;
      end if;
  else /*error*/;
  end case;
end op <;

op <=(a,b);
```

```
  case of
  (is_integer a and is_integer b):          $ integer less or equals
      return a<b or a=b;
  (is_real a and is_real b):                $ real less than or equals
      return a<b or a=b;
  (is_string a and is_string b):            $ string less than or equals
      return a<b or a=b;
  end case;
end op <=;

op >(a,b):
  case of
  (is_integer a and is_integer b):          $ integer greater than
      return b < a;
  (is_real a and is_real b):                $ real greater than
      return b < a;
  (is_string a and is_string b):            $ string greater than
      return b < a;
  else /*error*/;
  end case;
end op >;

op >=(a,b):
  case of
  (is_integer a and is_integer b):          $ integer greater or equals
      return a>b or a=b;
  (is_real a and is_real b):                $ real greater or equals
      return a>b or a=b;
  (is_string a and is_string b):            $ string greater or equals
      return a>b or a=b;
  else /*error*/;
  end case;
end op >=;

op abs(a);
  case of
  (is_integer a):                           $ absolute value of integer
      case of
      (a < 0):
      return -a;
      (a >= 0):
        return a;
      end case;
  (is_real a):                              $ absolute value of real
      case of
```

```
      (a < 0.0):
        return -a;
      (a >= 0.0):
        return a;
      end case;
  (is_string a):                          $ absolute value of string
      if #a /= 1 then
        /*error*/;
      else
        return /* integer code for a */;
      end if;
  else /*error*/;
  end case;
end op abs;

op and(a,b):                              $ logical and
  case of
  (a = FALSE);
                                          $ skip evaluation of b and
      return FALSE;
  (a = TRUE):
                                          $ evaluate b and do
      case of
      (b = TRUE):
        return TRUE;
      (b = FALSE):
        return FALSE;
      else /*error*/;
      end case;
     else /*error*/;
   end case;
end op and;

op arb(a);
   case of
   (is_set a):                            $ arbitrary element of set
      if a = { } then
        return om;
      else
        return /* arbitrary element from a */;
      end if;
   else /*error*/;
   end case;
end op arb;
```

```
op ceil(a);
  case of
  (is_real a):                          $ ceiling of real
     case of
     (a < 0.0):
       loop init x := 0; doing x -:= 1; do
         if float x < a then return x + 1;
       end loop;
     (a >= 0.0):
       loop init x := 0; doing x +:= 1; do
         if float x >= a then return x;
       end loop;
     end case;
  else /*error*/;
  end case;
end op ceil;

op div(a,b);
  case of
  (is_integer a and is_integer b):      $ integer division
     if b = 0 then /*error*/; end if;
     x := abs a;
     y := 0;
     (while x >= abs b)
       x -:= abs b;
       y +:= 1;
     end while;
     return y * sign a * sign b;
  else /*error*/;
  end case;
end op div;

op domain(a);
  case of
  (is_map a):                           $ domain of map
     return {x : [x,y] in a};
  else /*error*/;
  end case;
end op domain;

op even(a);
  case of
  (is_integer a):                       $ test even integer
```

```
      return a mod 2 = 0;
    else /*error*/;
    end case;
  end op even;

  op fix(a);
    case of
    (is_real a):                        $ convert real to integer
      assert exists x:=1,2.. | float x > abs a;
      return (x - 1) * sign a;
    else /*error*/;
    end case;
  end op fix;

  op float(a);
    case of
    (is_integer a):                     $ convert integer to real
      return /* result of converting a to real */;
    else /*error*/;
    end case;
  end op float;

  op floor(a);
    case of
    (is_real a):                        $ floor of real
      case of
      (a < 0.0):
        loop init x := 0; doing x -:= 1; do
          if float y <= a then return x;
        end loop;
      (a >= 0.0):
        loop init i := 0; doing i +:= 1; do
          if float i > a then return i - 1;
        end loop;
      end case;
    else /*error*/;
    end case;
  end op floor;

  op from(wr a, rw b);
    case of
    (is_set b):                         $ take from set
      a := arb b;
      if a /= om then b less:= a; end;
      return a;
```

```
    else /*error*/;
    end case;
  end op from;

  op fromb(wr a, rw b);
    case of
    (is_string b):                    $ take from start of string
       if b = '' then
          a := om;
       else
          a := b(1);
          b := b(2..);
       end if;
   (is_tuple b):                      $ take from start of tuple
       if b = [ ] then
          a := om;
       else
          a := b(1);
          b := b(2..);
       end if;
    else /*error*/;
  end op fromb;

  op frome(wr a, rw b);
    case of
    (is_tuple b):                     $ take from end of tuple
       if b = [ ] then
          a := om;
       else
          a := b(#b);
          b := b(1..#b-1);
       end if;
    (is_string b):                    $ take from end of string
       if b = '' then
          a := om;
       else
       a := b(#b);
       b := b(1..#b-1);
       end if;
       return a;
    else /*error*/;
    end case;
  end op frome;
```

```
op impl(a,b);
  case of
  (is_boolean a and is_boolean b):
     return (not a) or b;
  else /*error*/;
  end case;
end op impl;


op in(a,b);
  if a = om then /*error*/; end if;
  case of
  (is_set b):                          $ test element in set
     return exists x in b | x = a;
    (is_string a and is_string b):     $ test character in string
     if #a /= 1 then /*error*/; end;
     return exists x:=1..#b | a = b(x);
  (is_tuple b):                        $ test element in tuple
     return exists x:=1..#b | a = b(x);
  else /*error*/;
  end case;

op incs(a,b);
  case of
  (is_set a and is_set b):             $ set inclusion test
     return forall x in b | x in a;
  else /*error*/;
  end case;
end op incs;

op less(a,b);
  case of
  (is_set a):                          $ remove set element
     return {x in a | x /= b};
  else /*error*/;
      end op less;


op lessf(a,b);
  case of
  (is_map a):                        $ remove map element
     return {[x,y] in a | x /= b};
  else /*error*/;
```

```
    end case;
  end op lessf;

  op max(a,b);
    case of
    (is_integer a and is_integer b):     $ integer maximum
       case of
       (a>=b): return a;
       (a<=b): return b;
       end case;
   (is_real a and is_real b):            $ real maximum
       case of
       (a>=b): return a;
       (a<=b): return b;
       end case;
    else /*error*/;
    end case;
  end op max;

  op min(a,b);
    case of
    (is_integer a and is_integer b):     $ integer minimum
       case of
       (a<=b): return a;
       (a>=b): return b;
       end case;
    (is_real a and is_real b):           $ real minimum
       case of
       (a<=b): return a;
       (a>=b): return b;
       end case;
    else /*error*/;
    end case;
  end op min;

  op mod(a,b);
    case of
    (is_integer a and is_integer b):     $ integer modulus
       return a - b * (a div b);
    else /*error*/;
    end case;
  end op mod;

  op not(a);                             $ logical not
    case of
```

```
  (a=TRUE):
     return FALSE;
  (a=FALSE):
     return TRUE;
  else /*error*/;
  end case;
end op not;

op notin(a,b):
  case of
  (is_set b):                         $ test element not in set
     return not a in b;
  (is_string a and is_string b):      $ test character not in string
     return not a in b;
  (is_tuple b):                       $ test element not in tuple
     return not a in b;
  else /*error*/;
  end case;
end op notin;

op npow(a,b);
  case of
  (is_integer a and is_set b):        $ subsets of given size
     if a<0 then /*error*/; end;
     return {x in pow b | #x = a};
  (is_set a and is_integer b):        $ subsets of given size
   if b<0 then /*error*/; end;
     return {x in pow a | #x = b};
  else /*error*/;
  end case;
end op npow;


op odd(a);
  case of
  (is_integer a):                       $ test odd integer
     return a mod 2 /= 0;
  else /*error*/;
  end case;
end op odd;

op or(a,b);                             $ logical inclusive or
  case of
  (a = TRUE):
```

```
                                        $ skip evaluation of b and
     return TRUE;
   (a = FALSE):
                                        $ evaluate b and do
     case of
     (b = TRUE):
       return TRUE;
     (b = FALSE):
\         return FALSE;
     else /*error*/;
     end case;
   else /*error*/;
   end case;
  end op or;

  op pow(a);
   case of
   (is_set a):                          $ power set of set
     if a={ } then
       return {{ }};
     else
       x := arb a;
       y := pow (A less x);
     return y + {z with x : z in y};
     end;
   else /*error*/;
   end case;
  end op pow;

  op random(a);
   case of
   (is_integer a):                      $ random integer
     if a >= 0 then
       return fix random float (a+1);
    else
       return fix random float (a-1);
     end if;
   (is_real a):                         $ random real
     if a = 0.0 then
       return 0.0;
     elseif a < 0.0 then
       return - random -a;
     else
       return /* random real in the
         half open interval [0,a) */;
```

```
  (is_tuple a):                     $ random element from tuple
      if #a = 0 then
        return om;
      else
        return a(1 + random (#a-1));
      end if;
  (is_set a):                       $ random element from set
      if a = { } then
        return om;
      else
        return random [x: x in a];
      end if;
  else /*error*/;
  end case;
end op random;

op range(a);
  case of
  (is_map a):                       $ range of map
      return {y : [x,y] in a};
  else /*error*/;
  end case;
end op range;


op sign(a);
  case of
  (is_integer a):                                      $ integer sign
      case of
      (a<0): return -1;
      (a=0): return 0;
      (a>0): return +1;
      end case;
  (is_real a):                                         $ real sign
      case of
      (a<0.0): return -1;
      (a=0.0): return 0;
      (a>0.0): return +1;
      end case;
  else /*error*/;
  end case;
end op sign;

op str(a);
```

```
const dg = '0123456789',
      uc = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
      lc = 'abcdefghijklmnopqrstuvwxyz',
       q = '''';
case of
(is_boolean a):                                    $ boolean to string
    case a of
      ('TRUE'):  return '#T';
      ('FALSE'): return '#F';
    end case;
(is_integer a):                                    $ integer to string
    y := abs a;
    (until y=0)
      x +:= dg(y mod 10);
      y := y div 10;
    end;
    return if a < 0 then '-' + x else x end;
(is_string a):                                     $ string to string
    if a /= '' and a(1) in uc + lc and
         forall x in a(2..) | x in uc+lc+dg+'_' then
      return a;
    else
      return q +/ [if y=q then q+q else y end : y in a] + q;
    end if;
  (a=om):                                          $ undefined to string
    return '*';
(is_tuple a):                                      $ tuple to string
    x := '';
    (for y in a)
      x +:= ' ' + str y;
    end;
    return '[' + x(2..) + ']';
(is_set a):                                        $ set to string
    (for y in a)
      x +:= ' ' + str y;
    end;
    return '{' + X(2..) + '}';
end op str;

op susbset(a,b);
case of
(is_set a and is_set b):                           $ subset test
    return forall x in a | x in b;
else /*error*/;
end case;
```

```
end op subset;

op with(a,b);
 case of
 (is_set a):                                  $ add element to set
     return {x : x in [y : y in a] with b};
 (tuple(a)):                                  $ add element to tuple
     x := a;
     x(#x+1) := b;
     return x;
 else /*error*/;
 end case;
end op with;
```

CHAPTER 8

SYNTAX

This chapter contains a formal definition of the permitted syntax of SETL programs.

8.1. Syntax Rules

This section contains the syntax rules. A modified form of **bnf** is used. The syntactic types (or non-terminals) are strings of small letters, possibly including a underline. The possible alternatives, if there is more than one, are separated by the | character. Terminals are expressed in one of the following possible notations:

> A string of upper case letters represents a reserved word and can appear in either upper or lower case letters, or a mixture, in the final program.
>
> A special character, or sequence of special characters, represents itself as a terminal symbol. In the case of the characters [ ]{ } |, enclosing quotes are used to avoid confusion with the meta-syntactic use of these characters in the description.
>
> Special non-terminal types whose names begin with tok_ represent sets of terminals which are described in the following section.

The following notations are used to extend the normal **bnf** conventions:

> A sequence of items surrounded by brackets [ ] is used to indicate that the sequence of items is optional.
>
> A sequence of items surrounded by braces { } is used to indicate that the sequence of items may be repeated any number of times, or entirely omitted.
>
> The sequence [...] is used to represent the token sequences in an ender where the tokens are copied from the corresponding start construction.

In some cases, the **bnf** notation given is inadequate to state certain restrictions or additional rules. Special comments preceded by a dollar sign are given after the rule in question to provide the needed additional information.

A SETL program consists of a terminal string of type setl_prog, together with any referenced libraries, which are obtained as terminal strings of type lib_unit.

setl_prog	simple_prog | module_prog
simple_prog	**program** tok_nam; {lib_item} {decl} {stmt} {refine} {routine} **end** [**program** [...]] ;
module_prog	direc_unit prog_unit {module_unit}
direc_unit	**directory** tok_nam ; {lib_item} {decl} **program** tok_nam - tok_nam : dir_spec {**module** tok_nam - tok_nam : dir_spec } **end** [**directory** [...]] ;
lib_unit	**library** tok_nam ; {lib_item} {export_item} {decl} routine {routine} **end** [**library** [...]] ;
dir_spec	{dir_item} {decl_repr}
dir_item	read_item | write_item | imports_item | exports_item
read_item	**reads** tok_nam {, tok_nam} ;
write_item	**writes** tok_nam {, tok_nam} ;
imports_item	**imports** pspec {, pspec} ;
exports_item	**exports** pspec {, pspec} ;
lib_item	**libraries** tok_nam {, tok_nam} ;
pspec	tok_nam [(pspeca {, pspeca})] | tok_nam [({pspeca ,} pspeca (*)]

pspeca
: **rd** [tok_nam]
: **wr** [tok_nam]
: **rw** [tok_nam]
: [tok_nam]

prog_unit
: **program** tok_nam - tok_nam ;
: {lib_item}
: dir_spec
: {decl}
: {stmt}
: {refine}
: {routine}
: **end** [**program** [...]] ;

module_unit
: **module** tok_nam - tok_nam ;
: {lib_item}
: ir_spec
: {decl}
: routine
: {routine}
: **end** [**module** [...]] ;

decl
: **var** tok_nam {, tok_nam} [: BACK] ; |
: **const** declcon {, declcon} ; |
: **init** tok_nam := constant {, tok_nam := constant} ;
: decl_repr

declcon
: tok_nam [= constant]

constant
: [sign] tok_int | [sign] tok_rea |
: tok_str | tok_rea |
: "{" [constant {, constant}] "}" |
: "[" [constant {, constant}] "]" |
: "{" constant [, constant] .. constant "}" |
: "[" constant [, constant] .. constant "]"

sign
: +
: –

decl_repr
: **repr** repr {repr} **end** [**repr** [...]] ;

repr
: tok_nam {, tok_nam} : mode ; |
: **mode** tok_nam : mode ; |
: **base** tok_nam {, tok_nam} : mode ; |
: **plex base** tok_nam {, tok_nam} ;

mode
atom | **boolean** | **real** | **string** |
integer [(tok_int .. tok_int)] |
tuple [(mode) [(tok_int)]] |
tuple (mode , mode {, mode}) |
set [(mode)] |
basetype **set** (emode) |
maptype (mode) [mode] |
basetype maptype (emode) [mode] |
procedure [([mode {, mode}])] [mode] |
op (mode [, mode]) mode |
* | emode | tok_nam

emode
elmt tok_nam | tok_nam

basetype
local | **remote** | **sparse**

maptype
map | **mmap** | **smapa**

routine
procedure | opdef

procedure
proc tok_nam [arglist] ;
body
end [**proc** [...]] ;

arglist
(formal {, formal}) |
({formal ,} formal (*))

formal
[formtype] tok_nam

formtype
rd | **rw** | **wr**

opdef
op tok_ubo (tok_nam , tok_nam) ;
body
end [op [...]] ; |
op tok_uuo (tok_nam) ;
body
end [**op** [...]] ;

body
{decl}
{stmt}
{refine}

refine
tok_nam :: {stmt}

stmt	{tok_nam :} stmt_body
stmt_body	assert_stm | assign_stm | call_stm | case_stm | cont_stm | exit_stm | fail_stm | goto_stm | if_stm | loop_stm | pass_stm | quit_stm | return_stm | stop_stm | succeed_stm | yield_stm
assert_stm	**assert** expr;
assign_stm	lhs [tok_bop] := expr ;
call_stm	toknam [([expr {, expr}])] ;
case_stm	case_stm_t | case_stm_c case_stm_c
case_stmt	**case of** { (**expr** {, **expr**}) : {stmt} } [**else** {stmt}] **end** [**case**] ;
case_stm_c	**case** expr **of** { (constant {, constant}) : {stmt} } [**else** {stmt}] **end** [**case** [...]];
cont_stm	**continue** [...] ;
exit_stm	**exit** ;
fail_stm	**fail** ;
goto_stm	**goto** tok_nam ;

if_stm	**if** expr **then**
	  {stmt}
	  {elseif expr **then** {stmt}}
	  [**else** {stmt}]
	end [**if** [...]] ;

loop_stm	**loop** loopiter **do** {stmt} **end** [**loop** [...]] ; (loopiter) {stmt} **end** [...] ;
loopiter	**for** iterator | [init] [doing] [while] [step] [until] [term]
init	**init** {stmt}
doing	**doing** {stmt}
while	**while** expr
step	**step** {stmt}
until	**until** expr
term	**term** {stmt}
pass_stm	**pass** ;
quit_stm	**quit** [...] ;
return_stm	**return** [expr] ;
stop_stm	**stop** ;
succeed_stm	**succeed** ;
yield_stm	**yield** expr ;
iterator	iterelmt {, iterelmt} ["|" expr]
iterelmt	lhs **in** expr | lhs = tok_nam (lhs {, lhs}) | lhs = tok_nam { lhs {, lhs} }
lhs	tok_nam | [" lhst [, lhst} "]" |

```
                lhs ( expr {, expr} ) |
                lhs { expr {, expr} }

lhst            lhs | -

expr            tok_uop expr |
                tok_bop / expr |
                expr tok_bop expr |
                expr [tok.bop] :\(eq expr |
                expr tok_bop / expr |
                bopnd |
                lhs from lhs |
                lhs fromb lhs |
                lhs frome lhs |
                exists iterelmt {, iterelmt} "|" expr |
                notexists iterelmt {, iterelmt} "|" expr |
                forall iterelmt {, iterelmt} "|" expr |

bopnd           tok_int | tok_rea | tok_str | tok_nam |
                TRUE | FALSE | om | newat | eof | nargs | lev | ok | date | time |
                ( expr ) | case_expr | if_expr |
                "[" [ expr {, expr} ] "]" |
                "[" former "]" |
                "{" [ expr {, expr} ] "}" |
                "{" former "}" |
                bopnd ( expr {, expr} ) |
                bopnd ( expr .. [ expr ] ) |
                bopnd "{" expr {, expr} "}" |
                expr {stmt} end

former          expr : iterator |
                expr [, expr] .. expr |
                lhs in expr "|" expr

case_expr       case_expr_t | case_expr_c

case_expr_t     case of
                  casetb {, casetb} else expr
                end

casetb          ( expr {, expr} ) : expr

case_expr_e     case expr of
                  caseeb {, caseeb} else expr
                end
```

caseeb (constant {, constant}) : expr

if_expr **if** expr **then**
expr
{**elseif** expr **then** expr}
else expr
end

In these rules, there are certain token substitutions which are allowed. The following rules indicate the possible substitutions. The only reason that these are stated as separate rules is to avoid an unnecessary duplication of many rules which would otherwise be required.

In any rule where the set brackets "{" and "}" appear, the substitutions << and >> may be made, to effectively generate an extra alternative.

In any rule where the tuple brackets [and] appear, the substitutions (/ and /) may be made, to effectively generate an extra alternative.

The sequence of two dots .. which is used for various range constructs may always be replaced by ... instead, a sequence of three dots.

The character "|" may be replaced by the keyword **st** wherever it occurs.

8.2. Token Types

This section describes the token types which were referenced in the previous section.

tok_p binary operator
All defined binary operator names, including those defined using **op** definitions.

tok_int integer denotation
An integer denotation is a string of digits 0-9 of any length which fits on one line of the source program. It does not include any sign characters.

tok_nam identifer name
Formed as a sequence of letters, digits and the underline character, starting with a letter. Upper and lower case are equivalent. Excludes any of the reserved names corresponding to standard keyword and operator names, as listed in Appendix nn.

tok_rea real denotation
A real denotation is an optional string of digits, followed by a period, followed by a non-empty string of digits (note that there must be at least one digit after the decimal point). An optional exponent may follow consisting of the letter e (upper or lower case) followed by an optional sign (+ or −) followed by a non-empty digit string.

tok_str string denotation
A string denotation is an arbitrary sequence of characters enclosed using the character ’ (single

quote mark or apostrophe). If this character is to appear inside the string, it is written as two successive apostrophe characters. Note that a string denotation, like any other token, must fit on a single line. If a longer string is required, the effect is obtained by writing two or more successive string denotations on successive lines. Since the SETL language never allows two strings in adjacent positions, this can be unambiguously interpreted as a single long string constant.

tok_ubo user binary operator name
Consists of a period, immediately followed by a sequence of characters which meets the normal rules for an identifier (tok_nam). It is permissible for the name to be the same as an identifier name used elsewhere, the period serving to distinguish the uses.

tok_uop unary operator
Includes all standard unary operator names, together with any unary operators defined using an **op** definition.

tok_uuo user unary operator name
An identifier preceded by a period, exactly as for a binary operator.

TABLE OF CONTENTS