SETL Newsletter Number 49 August 10, 1971

Detailed Specifications of Certain SETL Operations

Henry Warren

This specification serves two purposes:

1. It is a starting point for a BALM implementation of a SETL-like language in which sets are accessed by means of a hashing scheme.

2. It is a detailed specification of the meaning of most current SETL operators, and a few other SETL constructions such as functional application.

The SETL features which have been specified are listed in the table of contents and in the index. The features which are not specified here are (1) all SETL statements (define, if, print, etc.), (2) iteration, (3) assignment and sinister functions, (4) use of square brackets in such constructions as f(x,[y]) and [s]+1, (5) use of the colon (as in s(1:3)), (6) hashing algorithms for atoms, and (7) the functions <u>hol</u>, <u>holl</u>, <u>is</u>, <u>log</u>, sin, cos, and <u>compile</u>.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

I. Introduction

Much of this specification has nothing to do with hashing, and that part will not be commented on here. A few words will be said about the hashing, however.

Each SETL object has a hash code associated with it. The hash code is treated here as a non-negative integer (rather than a bit string). Each non-empty set has a hash table associated with it. When an object is a member of a set, the object is placed in the hash table entry that corresponds to the object's hash code reduced modulo the size of the set's hash table. Thus, a set's hash table entry is a list of members of the set which map into the same hash table entry. This list is searched sequentially.

The primary goal of hashing is to provide a fast membership test. This is not so much because of the frequency of membership tests in typical SETL source programs, but is due to the fact that the membership test is basic to several other SETL set-operations, notably eq, with, and less. Since other operations (union, set forming, left-hand-side set expressions, etc.) are apt to be built on these, it is likely that a slow membership test will be disastrous for SETL.

If sets are represented as linear lists, then the membership test requires on the order of n operations to perform, where n is the size of the set. If sets are represented as trees, and the tree is kept "balanced", then the membership test requires $\sim$log n operations. If sets are represented using hash tables of size k, and clustering does not occur, then the membership test requires $\sim$max(n/k,1) operations, where k may be made as large as desired. In the implementation described here, k is varied roughly proportional to n, so the number of operations is independent of the size of the set. Thus, from this standpoint, hashing seems to be a reasonable way to deal with sets, particularly if one wishes to execute algorithms that involve large sets (on the order of 1000 members).

A secondary goal of hashing is to provide fast functional application when the function is represented by a set. That is, to evaluate f(x,y), we wish to quickly locate all tuples in f that begin with x,y. This is done by treating tuples in a special way when they are made members of sets. The tuple is broken up so that its first component is stored in the set's primary hash table, together with a reference to the set of all tails of tuples that are in the set and that have the same first component. This scheme is described in more detail in the following section.

There are many details in this specification which should be changed when coding it in some particular language for execution on some particular machine. For example, the modulus used in connection with hashing might be changed to be an integral power of two. More significantly, the hash code of objects (particularly arithmetic atoms) might be calculated only when needed rather than always calculated and saved. As another example, one might reduce the size of a hash table only at garbage collection time, to avoid the check in every <u>less</u> operation.

Finally, it is pointed out that the algorithms are geared to what are believed to be the common cases and for large sets. Sometimes these specifications result in grotesque constructions. For example, to remove a tuple from a set, it must be removed component by component, and put back together again by concatenation. As another example, a set containing a single 100-tuple requires 99 hash tables for its representation.

II. Internal representation of SETL objects

A SETL object is represented by a triple (BALM list or vector) of the form

$\langle$type, hash-code, value$\rangle$.

"type" is an integer whose value is (symbolically) integer, real, bstring, cstring, label, blank, subroutine, function, tupl, or set.

"hash-code" is an integer on the order of 10 to 40 bits long.

"value" is:

(1) The value if the object is an atom.

(2) A pair of the form $\langle$#elements, tuple$\rangle$ if a tupl, where "tuple" is an n-tuple (e.g. BALM vector).

(3) A 4-tuple of the form $\langle$#elements, ht size, ht load, hash table$\rangle$ if a set. Here the first three components are integers, and "hash-table" is a tuple (BALM vector) of length "ht size", whose components are n-tuples (BALM lists or vectors) of elements of the set being coded. These elements are triples as above.

For example, the SETL integer "2" might be represented as:

$\langle$ 1,84321683,2 $\rangle$.

The SETL set $\{1,2,3\}$ might be:

$\langle$11,13118432,$\langle$3,8,3 nl,nl,nl,
$\langle\langle$1,74321876,1$\rangle$,$\langle$1,84321683,2$\rangle\rangle$,
nl,nl,$\langle$1,45328471,3$\rangle$,nl $\rangle\rangle$

The "ht load" component is the number of items attached to the hash table. It is used to determine when a hash table should be expanded or contracted. For many sets, "ht load" is equal to "# elements" (the number of elements in the set). However, for sets containing tuples of size three and up, "ht load" may be considerably less than "# elements". This is because of the fact that tuples of size three and larger are broken up, when added to a set, in a manner that facilitates locating subsets of such tuples all of which have the same first few components.

The picture on the following page illustrates this "breaking up" process. The hash table entries for a set s are lists of

(1) non-tuple members of s,

(2) zero-, one-, and two-tuple members of s, and

(3) SETL triples of the form $\langle \underline{tupl}$, hash_code(a), $\langle 3, \langle a, Sa, 0 \rangle\rangle\rangle$

where
"a" is the first component of an n-tuple ($n \geq 3$) member of s, Sa is a set containing the tails of these n-tuples that begin with a, and the zero is included to make the list entry a triple.[1]

To evaluate f{a,b,c,d}, the set f must be searched for members having one of the following forms, where x is either an object or the first component of a tuple.

$\langle a, \langle b, \langle c, x \rangle\rangle\rangle$

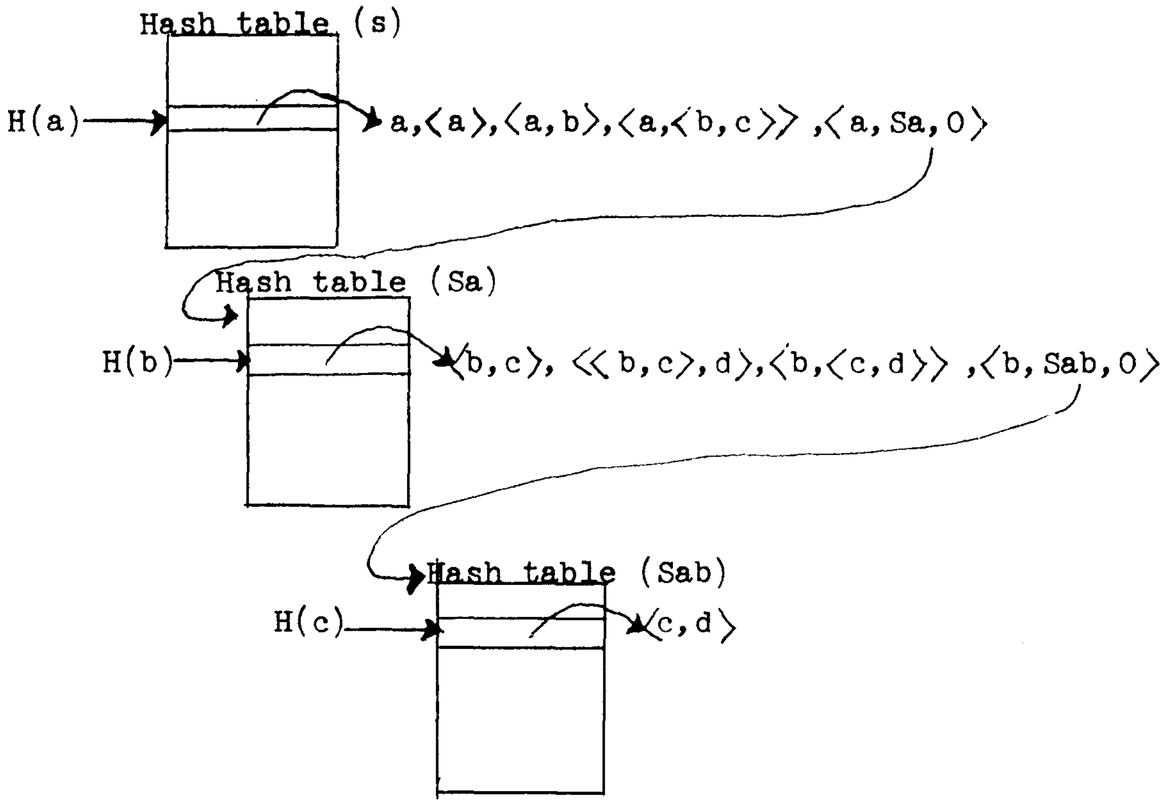
$\langle a, \langle b, c, x \rangle\rangle$

$\langle a, b, \langle c, x \rangle$

$\langle a, b, c, x \rangle$

[1] Note that the number of elements of an n-tuple is not explicitly maintained if $n \geq 3$ and the tuple is a member of a set. Note also that the subject of sparse tuples is not addressed.

$s=\{a,\langle a\rangle,\langle a,b\rangle,\langle a,\langle b,c\rangle\rangle,\langle a,b,c\rangle,\langle a,\langle b,c\rangle,d\rangle$
$\langle a,b,\langle c,d\rangle\rangle,\langle a,b,c,d\rangle\}$


Hash table (s)
H(a)
a,⟨a⟩,⟨a,b⟩,⟨a,⟨b,c⟩⟩ ,⟨a, Sa, 0⟩
Hash table (Sa)
H(b)
⟨b,c⟩, ⟨⟨b,c⟩,d⟩,⟨b,⟨c,d⟩⟩ ,⟨b, Sab, 0⟩
Hash table (Sab)
H(c)
⟨c,d⟩

III. Special conventions concerning Ω

1. type Ω is Ω .
2. atom Ω is t.
3. $\#\Omega \Rightarrow$ error exit.
4. Ω eq Ω is t, Ω ne Ω is f.
5. $\Omega \in s$, Ω n$\in s \Rightarrow$ error exit
6. $x \in \Omega$, x n$\in \Omega \Rightarrow$ error exit
7. Ω with $x \Rightarrow$ error exit

Generated or Propagated by

1. f(x), if x n$\in$ hd f (but f(Ω) and Ω(x)$\Rightarrow$ error exit).
2. type Ω
3. x = Ω
4. hd Ω, hd⟨ ⟩, hd⟨Ω,...⟩
5. tl Ω, tl⟨ ⟩, tl⟨x⟩
6. ∃nl, ∃nult, ∃nulc, ∃nulb
7. tupl(n) if n≤0 or >#tupl or the n-th component is missing.

8.	s with $\Omega \Rightarrow$ error exit	8.	string(n) if $n \le 0$ or $> \#$string.
9.	Ω less x $\Rightarrow$ error exit	9.	$\exists [x] \in s \mid C(x)$, no such x found.
10.	s less $\Omega \Rightarrow$ error exit	10.	f[nl], f[nult], f[nulc], f[nulb].
11.	$\ni \Omega \Rightarrow$ error exit	11.	[op:$\forall x \in s$]..., s is nl, nulb, nulc, or nult.
12.	$\Omega\{x\}$, $f\{\Omega\} \Rightarrow$ error exit	12.	npow(n,s), $n > \# s$.
13.	$\Omega\{x_1,\dots,x_n\}$, $f\{x_1,\dots,\Omega,\dots,x_n\}$ $\Rightarrow$ error exit	13.	[op:$\forall x \in s$]exp; s is nl, nulb, nulc, or nult.
14.	f(x) and f[x] are similar to (12) and (13) above.	14.	$\exists [x] \in s \mid C(x)$, no such x exists
15.	pair Ω is f.	15.	random nl

1. Arguments are not checked unless they are actually used. For example, in "x $\in$ s", if s is nl, a value of f is returned even if x is Ω. Similarly, in $f\{x_1,\dots,x_n\}$, if no tuple in f begins with x_1, a nl result is returned and $x_2,\dots,x_n$ are not examined for legality.

2. The rules concerning Ω are usually derived from the following principle. If a set or tuple is referenced in a way that requests a member having certain properties, and no such member exists, then the value of the expression is Ω. On the other hand, execution is generally terminated if a SETL operation is used with incorrect data types, or "obviously" incorrect values (including Ω), such as division by 0.

Thus the main sources of Ω are certain retrieval operations (f(x), $\ni$s, hd, tupl(n), string(n)) when the item to be retrieved is not defined (i.e., does not exist).

An essential exception to the above is that Ω is allowed to enter into the eq and ne operations, so that it may be tested for. As a convenience, Ω is also allowed to be the argument of the SETL type, atom, and pair functions, as the use of these indicates that there is some doubt about the status of a SETL object. Also as a convenience, Ω is allowed in hd, tl, and assignment operations.

3. Ω is allowed to be a component of a tuple, but it is not allowed to be a member of a set. As a component of a tuple, it serves as a "place marker". Let

$t1=\langle 1,\Omega,3,\Omega\rangle$

and $t2=\langle 1,\Omega,3\rangle$.

Then $\#t1=4$, $\#t2=3$,

$t1(0)=t1(2)=t1(4)=t1(5)=\Omega$.

$t1(4:)=\langle\Omega\rangle$, $t1(5:)=\Omega$,

$t1(4:2)=t1(5:2)=\Omega$, t1 eq t2 is false.

IV. Detailed specifications

A. Miscellaneous utility routines and parameters

These routines take object-language objects (triples) as arguments.

```
definef type(a); return a(1); end;
definef hash_code(a); return a(2); end;
definef value(a); return a(3); end;

definef nelt(a); /* use for sets or tuples only. */
return a(3)(1); end;

definef tuple(a); /*  use for tuples only.  */
return a(3)(2); end;

definef ht_size(a);  /*  use for unordered sets only.  */
return a(3)(2); end;

definef ht_load(a); /*  use for unordered sets only.  */
return a(3)(3); end;
```

```
definef hash-table(a); /* use for unordered sets only. */
return a(3)(4);end;

definef triple(x);
return(type(x) eq tupl and nelt(x) eq 3);
end triple;
```

These routines take meta-language objects (integers, t, f, etc.) as arguments.

```
definef repint(x); return<int,hashint(x),x>; end;
definef repreal(x);return<real,hashreal(x),x>;end;
definef repbstr(x);return<bstring,hashbstr(x),x>;end;
definef repcstr(x);return<cstring,hashcstr(x),x>;end;
definef replabel(x);return<label,hashlabel(x),x>;end;
definef repblank(x);return<blank,hashblank(x),x>;end;
definef repsubr(x);return<subroutine,hashsubr(x),x>;end;
definef repfun(x);return<function,hashfun(x),x>;end;

/* It is assumed that the hash functions of atoms, referenced
above, are coded in a language of lower level than SETL.[2] */

newat_number=0; /* Initialization for newat function. */
```

B. Error routines

```
define error_type(message);
print 'Invalid data type for the SETL operation' + message;
exit;
end error_type;
```

[2] To code them in SETL would require the addition of something like the UNSPEC function of PL/I.

```
define error_impl(message);
/* Exit for detected implementation errors. */
print 'Implementation error.' + message;
exit;
end error_impl;

define error_value(message);
print 'Invalid data value for the SETL operation' + message;
exit;
end error_value;
```

C. Hashing routines and parameters

```
definef compressed_hash_code (max_value, hash_code);
return (hash_code//max_value+1);
end compressed_hash_code;
hash_null_set=12345;
hash_null_tupl=31416;
modulus=100000; /* 100000 used so code will fit in 17 bits. */
hash_null_cstring=27182;
hash_null_bstring=66256;

definef hash_set_with(s,a);
return (hash_code(s)+hash_code(a))//modulus;
end hash_set_with;

definef hash_set_less(s,a);
return(hash_code(s)-hash_code(a)+modulus)//modulus;
end hash_set_less;
```

```
smallest_hash_table_size=4;
max_density=2.0;
min_density=0.5;

define make_larger_hash_table(s);
new_size=ht_size(s)* 2;
(1≤∀j≤new_size)new_ht(j)=nult;;/* clear new hash table. */
(1≤∀i≤ht_size(s))
      (∀x∈ hash_table(s)(i))
            index=compressed_hash_code (new_size,hash_code(x));
            new_ht(index)=<x>+new_ht(index);
            end ∀x;
      end  ∀i;
ht_size(s)=new_size;       /*   (copy)   */
hash_table(s)=new_ht;      /*   (copy)   */
return;
end make_larger_hash_table;

define make_smaller_hash_table(s);
new_size=ht_size(s)/2;
(1≤ ∀j≤new_size)
      new_ht(j)=hash_table(s)(j)+hash_table(s)(j+new_size);
      end  ∀j;
ht_size(s)=new_size;       /*   (copy)  */
hash_table(s)=new_ht;      /*   (copy)  */
return;
end make_smaller_hash_table;

define check_for_larger_hash_table(s);
if ht_load(s) gt (max_density * ht_size(s))
      then make_larger_hash_table(s);;
return;
end check_for_larger_hash_table;
```

```
define check_for_smaller_hash_table(s);
if ht_load(s) ge (min_density*ht_size(s))  then return;;
if ht_size(s) gt smallest_hash_table_size then
             make_smaller_hash_table(s);return;end if;
if nelt(s) eq 0 then s=nullset;
return;
end check_for_smaller_hash_table;

define make_same_size_hash_tables(a,b);
(while hash_table_size(a) ne hash_table_size(b))
      if hash_table_size(a) gt hash_table_size(b) then
      make_smaller_hash_table(a);else make_smaller_hash_table(b);
              end if;
      end while;
return;
end make_same_size_hash_tables;
```

D. Special SETL constants

```
undef=<0,0,0>;
nullset=<set,hash_null_set,<0,0,0,0>>;
nulltupl=<tupl, hash_null_tupl,<0,< >>>;
nullcstring=<cstring,hash_null_cstring,nulc>;
nullbstring=<bstring,hash_null_bstring,nulb>;
false=repbstr(0b);
true=repbstr(1b);
```

E. SETL operators

```
definef atom(x);  /*  Specifies atom x.   */
if atom_tf(x) then return true; else return false;;
end atom;

definef atom_tf(x);
if type(x) eq set then return f;;
if type(x) eq tupl then return f;;
return t;  /*  Note that Ω is considered to be an atom.  */
end atom_tf;

definef size(a); /*  Specifies # a.   */
if type(a) eq set then return repint(nelt(a));;
if type(a) eq tupl then return repint(nelt(a));;
if type(a) eq cstring then return repint(#value(a));;
if type(a) eq bstring then return repint( # value(a));;
error_type ('#a,a not a set, tuple, cstring, or bstring.');
end size;

definef equal(a,b);  /*  Specifies a eq b.  */
if equal_tf(a,b) then return true; else return false;;
end equal;
```

```
definef not_equal(a,b); /* Specifies a ne b. */
if equal_tf(a,b) then return false; else return true;;
end not_equal;

definef equal_tf(a,b);
/* Two SETL objects are considered "equal" if they have
the same type and the same value, or both objects are Ω .
Thus, a real is never equal to an integer, a bstring is never
equal to a cstring, a tupl is never equal to a set, etc. Any
two SETL objects may be compared, and the result is always either
true or false. In particular, ( Ω eq Ω ) is true, and
( Ω eq anything else) is false.
      To be equal, strings, tuples, and sets must have the
same number of elements. Strings and tuples must have corres-
ponding elements equal. For sets, it is sufficient that every
element of each set be contained in the other. */
/* A frequent use of this function is by the membership test for
sets, in which the objects being compared are usually not equal.
This use usually results in an exit from the first line below. */
if hash_code(a) ne hash_code(b) then return f;;

/* Objects are probably equal, as they have the same hash
codes. Make a quick test on the object's type and value fields. */
if type(a) ne type(b) then return f;;
if value(a) eq value(b) then return t;;
```

```
/* The cases where either a= Ω or b= Ω  have been disposed
of, and it is known that a and b are of the same type and have
the same hash codes, but they have different value fields.  If
they are atoms, they are unequal.  If they are sets or tuples,
more testing, of a recursive nature, is required.  */
if atom_tf(a) then return f;;

/* We have either tuples or sets which are probably equal
(as they have the same hash codes).  It is expedient to compare
the sizes of the sets or tuples, partly because a few cases are
immediately disposed of, but mainly because the tuple and set
comparison algorithms become simpler.  */
if nelt(a) ne nelt(b)  then return f;;

/*  Note that the cases where either a or b is null have now
definitely been disposed of (this observation is crucial for the
correctness of the set equality test below).
if type(a) eq tupl then
      /*  compare corresponding elements (recursively)   */
      tupa=tuple(a);
      tupb=tuple(b);
      (1≤ ∀i≤ #tupa) if not equal_tf(tupa(i),tupb(i))
                     then return f; end if;
      end ∀i;
      return t;
      end if  /*  tuple case  */;

/* The only type remaining is the set.  They are probably
equal, as they are of equal size and have the same hash codes.
Therefore, an exhaustive membership test will probably have to
be done.
```

The algorithm below first forces the sets to have equal-sized hash tables. This is done to shorten the look-up time to verify that each element of a is contained in b. Given two equal sets which have different size hash tables (presumably because one grew larger and the other grew smaller), this feature of the algorithm probably will not significantly alter the execution time of the comparison. However, it is anticipated that if a set enters into a comparison operation once, then it (or a close approximation to it) probably will again, and thus the forcing of equal-sized hash tables may have significant long-term benefit. It is of help only for fairly dense hash tables, in particular, when the size of the hash table is smaller than the size of the set. */

```
make_same_size_hash_tables(a,b)
n=ht_size(a);
(1≤ ∀i≤n) tupa=hash_table(a)(i);
          tupb=hash_table(b)(i);
          if # tupa ne # tupb then return f;;
          (∀ x ∈ tupa) if not(∃y ∈ tupb | equal_tf(x,y))
                         then return f; end if;
          end ∀ x;
          end ∀ i;
return t;
end equal_tf;

definef el(x,s); /*  Specifies x ∈ s.  */
if el_tf(x,s) then return true; else return false;;
end el;
```

```
definef not_el(x,s); /* Specifies x n∈s. */
if el_tf(x,s) then return false; else return true;;
end not_el;

definef el_tf(x,s);
/*  The SETL membership test results in an error exit if
either x is undefined, or if s is a non-string atom (including
Ω), or if s is a string and x is not a string of the same
type.  In all other cases, either t or f is returned. */
if x eq undef then error_type('el(x,s),x is the undefined atom.');;
if type(s) eq set then return el_tf_set(x,s);;
if type(s) eq tupl then return el_tf_tuple(x,s);;
if type(s) eq cstring then return el_tf_cstring(x,s);;
if type(s) eq bstring then return el_tf_bstring(x,s);;
error_type ('el(x,s),s is neither a set, tuple,cstring,nor bstring.');
end el_tf;

definef el_tf_set(x,s);
if s eq nullset then return f;;
index=compressed_hash_code(ht_size(s),hash_code(x));
if type(x) eq tupl and nelt(x) ge 3 then
      hx=head(x);
      tx=tail(x);
      (∀ y∈hash_table(s)(index))
            if triple(y) and equal_tf(hx,tuple(y)(1))
            then return el_tf_set(tx,tuple(y)(2)); end if;
            end ∀y;
      return f;
      end if type;
(∀y∈hash_table(s)(index))
      if equal_tf(x,y) then return t; end if;
      end ∀y;
return f;
end el_tf_set;
```

```
definef el_tf_tuple(x,s);
if s eq nulltupl then return f;;
(∀ y ∈ tuple(s))
        if equal_tf(x,y) then return t; end if;
        end ∀y;
return f;
end el_tf_tuple;

definef el_tf_cstring(x,s);
if type(x) ne cstring then
        error_type('el(x,s),s is a cstring but x is
        not a cstring.'); end if;
vx=value(x);
(∀ c ∈ value(s))
        if vx eq c then return t; end if;
        end ∀c;
return f; /* (Return here if s is nulc, or if #vx≠1.) */
end el_tf_cstring;

definef el_tf_bstring(x,s);
if type(x) ne bstring then
        error-type ('el(x,s), s is a bstring but x is
        not a bstring.'); end if;
vx=value(x);
( ∀ b ∈ value(s))
        if vx eq b then return t; end if;
        end  ∀b;
return f; /* (Return here if s is nulb or if # vx≠1.) */
end el_tf_bstring;
```

```
definef head(t); /*  Specifies hd t.  */
if t eq undef or t eq nulltupl then return undef;;
if type(t) ne tupl then error_type('head(t), t is defined
              but not a tuple.');;
/*  t is a tuple with one or more components.  */
return tuple(t)(1);
end head;

definef tail(t); /*  Specifies tl t.  */
if t eq undef then return undef;;
if type(t) ne tupl then error_type('tail(t),t is defined
              but not a tuple.');;
/*  t is a tuple.  */
if nelt(t) le 1 then return undef;;
/*  t is a tuple with two or more components.  */
new_l=nelt(t)-1;
new_t(1:new_l)=t(2:); /*  copy.  */
return <tupl,hash_code(new_t(1)),<new_l,new_t>>;
end tail;
```

```
definef with(s,a); /* Specifies s with a. */
result=s; /* copy */
augment(result,a);
return result;
end with;

definef less(s,a); /* specifies s less a. */
result=s; /* copy */
diminish(result,a);
return result;
end less;

define augment(s,a);
/* This subroutine augments set s by adding element a to it;
it is equivalent to s=s with a. */
if type(s) eq set then augment_set(s,a); return;
else error_type('augment(s,a), or with(s,a), s is not a set.');end if;
end augment;

define augment_set(s,a);
if a ne undef then augment_set_aok(s,a);return;
else error_type('augment(s,a),or with (s,a),
      a is the undefined atom.'); end if;
end augment_set;

define augment_set_aok(s,a);
if type(a) eq tupl then augment_set_tuple(s,a);
else augment_set_simply(s,a); end if;
return;
end augment_set_aok;
```

```
define augment_set_tuple(s,t);
if nelt(t) le 2 then augment_set_simply(s,t);return; end if;
ht=head(t);
tt=tail(t);
if s ne nullset then
      index=compressed_hash_code(ht_size(s),hash_code(ht));
      /* Search for a triple beginning with ht. */
      (∀x∈hash_table(s)(index))
           if triple(x) and equal_tf(tuple(x)(1),ht) then
                /* Found it. */
                old_size=nelt(tuple(x)(2));
                augment_set_tuple(tuple(x)(2),tt);
                if nelt(tuple(x)(2)) ne old_size then nelt(s)=
                        nelt(s)+1
                return; end if;
           end ∀x;
      end if s;
/* There is no triple beginning with ht. Make one. */
trip=<tupl, hash_code(ht),<3,<ht,set_with_only(tt),0>>> ;
augment_set_simply(s,trip);
return;
end augment_set_tuple;

define augment_set_simply(s,a);
if s eq nullset then
      (1≤∀i≤smallest_size_hash_table)ht(i)=nult;;
      index=compressed_hash_code(smallest_size_hash_table,
                hash_code(a));
      s=<set,hash_set_with(nullset,a),
                <1,smallest_size_hash_table,1,ht >> ;
      return;
      end if;
```

```
index=compressed_hash_code(ht_size(s),hash_code(a));
subset_tuple=hash__table(s)(index);
(∀x∈subset_tuple) if equal_tf(x,a) then return; end if; end ∀x;
hash_table(s)(index)=<a>+subset_tuple;
hash_code(s)=hash_set_with(s,a);
nelt(s)=nelt(s)+1;
ht_load(s)=ht_load(s)+1;
check_for_larger_hash_table(s);
return;
end augment_set_simply;

definef set_with_only(a);
/* Allocate and initialize a hash table. */
(1≤∀i≤smallest_size_hash_table) ht(i)=nult; end ∀i;

index=compressed_hash_code(smallest_size_hash_table,hash_code(a));
if type(a) eq tupl and nelt(a) ge 3 then
      s=set_with_only(tail(a));
      ht(index)=<<tupl,hash_code(a),<3,<head(a),s,0>>>>;
else  /* (usual case)   */
      ht(index)=<a>; /* (BALM list containing only a) */
end if;
return <set,hash_set_with(nullset,a),
                  <1,smallest_size_hash_table,1,ht>>;
end set_with_only;

definef tuple_with_only(a);
/* Used for the SETL expression <a>. */
/* Note that it is permissible to form <Ω> .  */
return<tupl, hash_code(a),<1,<a>>>; /* (BALM vectors or lists) */
end tuple_with_only;
```

```
define diminish(s,a);
/* This subroutine diminishes the set s by deleting element a
from it; it is equivalent to s=s less a. */
if type(s) eq set then diminish_set(s,a); return;
else error_type('diminish(s,a), or less(s,a), s is not a set.');
end if; end diminish;

define diminish_set(s,a);
if a ne undef then diminish_set_aok(s,a); return;
else error_type('diminish(s,a), or less(s,a), a is the
      undefined atom.'); end if;
end diminish_set;

define diminish_set_aok(s,a);
if type(s) eq tupl then diminish_set_tuple(s,a);
else diminish_set_simply(s,a); end if;
return;
end diminish_set_aok;

define diminish_set_tuple(s,t);
if nelt(t) le 2 then diminish_set_simply(s,t); return; end if;
if s eq nullset then return; end if;
/*  s is a non-empty set and t is a tuple with three or more
components.  t is removed by recursively searching s for the
set containing the last two components of t.  When((and if) found,
this pair is removed from its set.  If this removal causes the
set containing the pair to become empty, then the triple in
the next higher order set is removed.  Otherwise, the triple
is left in, but the number signifying the number of elements
in the higher order set is decremented.  It is not necessary
to adjust the hash code of the higher order sets unless a triple
is removed.  */
```

```
ht=head(t);
tt=tail(t);
index=compressed_hash_code(ht_size(s),hash_code(ht));
($\forall$x $\in$hash_table(s)(index))
      if triple(x) and equal_tf(head(x),ht) then
          /* Found a triple beginning with t(1). */
          old_size=nelt(tuple(x)(2)); /* Remember the size of
         set x(2). */
          diminish_set_tuple(x(2),tt);
          if nelt(tuple(x)(2)) ne old_size then
          /* tt was removed from tuple(x)(2). */
          if nelt(x(2)) eq 0 then diminish_set_simply(s,x);
          else nelt(s)=nelt(s)-1; end if nelt;
      end if nelt;
      return; /* Stop $\forall$x scan. */
      end if type;
      end $\forall$x;
return; /* t n $\in$s */ end diminish_set_tuple;

define diminish_set_simply(s,a);
if s eq nullset then return; end if;
index=compressed_hash_code(ht_size(s),hash_code(a));
subset_tuple=hash_table(s)(index);
(1$\leq$ $\forall$i$\leq$#subset_tuple)
      if equal_tf(subset_tuple(i),a) then
          /* remove element a from the set. */
          hash_table(s)(index)=subset_tuple(1:i-1)+subset_tuple(i+1:);
          hash_code(s)=hash_set_less(s,a);
          nelt(s)=nelt(s)-1;
          ht_load(s)=ht_load(s)-1;
          check_for_smaller_hash_table(s);
          return; /* Stop $\forall$i scan. */
      end if equal_tf;
end  ∀i;
return; /*  a n∈ s  */
end diminish_set_simply;

definef  lesf(s,a); /*  Specifies s lesf a.  */
result=s; /*  copy  */
diminishf(result,a)
return result;
end lesf;

define diminishf(s,a)
/*  This subroutine diminishes the set s by deleting all
members that are tuples beginning with "a"; it is equivalent
to s=s lesf a. */
if type(s) eq set then diminishf_set(s,a); return;
else error_type('diminishf(s,a), or lesf(s,a), s is not a set.');
end if;
end diminishf;

define diminishf_set(s,a);
/*  Note:  a=undef is ok, as putting Ω in a tuple is allowed.*/
if s eq nullset then return; end if;
index=compressed_hash_code(ht_size(s),hash_code(a));
/*  Search for tuples beginning with a, and delete them.  */
subset_tuple=hash_table(s)(index);
(1≤∀i≤#subset_tuple)
      x=subset_tuple(i);
      if type(x) eq tupl and nelt(x) ge 1 and equal_tf(hd tuple(x),a)
           then /*  Found one.  Delete it.  */
           number_deleted=if nelt(x) eq 3 then nelt(tuple(x)(2))else 1
           hash_table(s)(index)=subset_tuple(1:i-1)+
                                     subset_tuple(i+1:);
           hash_code(s)=hash_set_less(s,x);
```

```
          nelt(s)=nelt(s)-number_deleted;
          ht_load(s)=ht_load(s)-1;
          check_for_smaller_hash_table(s);
      end if type;
      end  ∀i;
return;
end diminishf_set;

definef arb(s); /*  Specifies ∋s.  */
if type(s) eq set then return arb_set(s);;
if type(s) eq tupl then return head(s);;
if type(s) eq cstring then return repcstr(value(s)(1));;
if type(s) eq bstring then return repbstr(value(s)(1));;
error_type('arb(s), s is not a set, tuple, or string.');
end arb;

definef arb_set(s);
if s eq nullset then return undef; end if;
x=arb_simply(s);
if not triple(x) then return x;end if;
/*   x is the special triple for tuples in sets, that is, x
has the form <tupl, hash_code,<3,<a,Sa,0>>>.      */
y=arb_set(tuple(x)(2));
tuple(x)=<head(x)>  + tuple(y);
nelt(x)=1 + nelt(y);
return x;
end arb_set;

definef arb_simply(s);
(1<∀i<ht_size(s))
      if hash_table(s)(i) ne nult then
            return hd hash_table(s)(i); /*   copy.  */ end if;
      end ∀i;
```

```
error_impl('A set ne nullset has no used entries
          in its hash table.');
end arb_simply;

define augment_union(s,p);
/*  Augment set s with a copy of each of the members of set p.
Equivalent to s=s u p.  */
if type(s) ne set or type(p) ne set then
      error_type('augment_union(s,p),either s or p is not a set.');
else augment_union_args_ok(s,p); end if;
return;
end augment_union;

define augment_union_args_ok(s,p); /*  Used by f{x}.  */
/*  First set "ss" (small set)=smaller of s and p, and
      s=larger of s and p.  */
if nelt(s) ge nelt(p) then ss=p; /*  Reference assignment is ok. */
else ss=s;    /*  copy  */
      s=p;    /*  copy  */  end if;
/*  Now put all elements of ss in s.  */
(1≤ ∀i≤ht_size(ss)) /*  If ss is null then ht_size(ss) is zero.  */
      (∀x ε hash_table(ss)(i))
            if not triple(x) then augment_set_simply(s,x);
            else /*  x is a triple of the form <a,S1a,O>. Must see
                 if it has a counterpart <a,S2a,O> in set s.  */
                 index=compressed_hash_code(ht_size(s),hash_code(x));
                 (∀y ε hash_table(s)(index))
                     if triple(y) and equal_tf(tuple(y)(1),
                           tuple(x)(1)) then
                        augment_union_args_ok(tuple(x)(2),
                           tuple(y)(2));
                        continue ∀x;/* (stop search on y) */
                           end if type;
```

```
                    end ∀y;
                        augment_set_simply(s,x);
                    end if type;
                    end ∀x;
               end ∀i;
return; end augment_union_args_ok;

definef plus(a,b); /* Specifies a+b. */
ta=type(a); va=value(a);
tb=type(b); vb=value(b);
if ta ne tb then
      if (ta eq int and tb eq real) or
         (ta eq real and tb eq int) then
          return repreal(va+vb);
      else error_type('plus(a,b), a and b are of dissimilar
      types other than real, integer.');;;
/* ta=tb */
if ta eq int then return repint(va+vb);;
if ta eq bstring then return repbstr(va+vb);;
if ta eq cstring then return repcstr(va+vb);;
if ta eq real then return repreal(va+vb);;
if ta eq set then return union_args_ok(a,b);;
if ta eq tupl then return
     <tupl, hash_code(a),<nelt(a)+nelt(b),tuple(a)+tuple(b)>>;;
error type('plus(a,b),a and b are both not an integer, string,
       real, set, or tuple.');
end plus;

definef union(a,b); /* Specifies a u b. */
if type(a) eq set and type(b) eq set then
      return union_args_ok(a,b);
else error_type('union(a,b), a and b are not both sets.');;
end union;
```

```
definef union_args_ok(a,b);
if nelt(a) ge nelt(b) then
      result=a; /* copy */
      augment_union_args_ok(result,b);
else result=b; /* copy */
      augment_union_args_ok(result,a);
end if nelt;
return result;
end union_args_ok;

definef minus(a,b); /* Specifies a-b. */
ta=type(a); va=value(a);
tb=type(b); vb=value(b);
if ta ne tb then
      if (ta eq int and tb eq real) or
         (ta eq real and tb eq int) then
          return repint(va-vb);
      else error_type('minus(a,b), a and b are of dissimilar
          types other than real, integer.');;;
/* ta=tb */
if ta eq int then return repint(va-vb);;
if ta eq bstring then return and (a,not(b));;
if ta eq real then return repreal(va-vb);;
if ta eq set then
      result=a; /* copy */
      p=nult;
      next=nextelt(b,p);
      (while next ne undef)
           diminish_set_aok(result, next);
           next=nextelt(b,p);
           end while;
      return result; end if ta;
```

```
error_type('minus(a,b),a and b are both not an integer,
      bstring, real, or set.'); end minus;

definef times(a,b); /*   Specifies a * b.   */
ta=type(a);  va=value(a);
tb=type(b);  vb=value(b);
if ta eq int then
      if tb eq int then return repint(va * vb);;
      if tb eq real then return repreal(va * vb);;
      if tb eq bstring then return rep_s(a,b);;
      if tb eq cstring then return rep_s(a,b);;
      error_type('times(a,b), a an integer, b is neither an
          integer, real, nor string.'); end if ta;
if ta eq real then
      if tb eq real or tb eq int then return repreal(va * vb);;
      error_type('times(a,b), a real, b is neither real nor an
          integer.');
      end if ta;
if ta eq set then
      if tb eq set then return intersect(a,b);;
      error_type('times(a,b),a is a set but b is not.');
      end if ta;
if ta eq cstring or ta eq bstring then
      if tb eq int then return (hash_code(a)//vb)+1;;
      end if ta;
error_type('times(a,b), a is neither an integer, real, set,
      or string.');
end times;
```

```
definef rep_s(n,s); /* n * s for integer n, string s. */
/* Note: s may be either a character or a boolean string. */
vn=value(n);
if vn lt 0 then error_value('n * s, string s, n<0.');;
vs=value(s);
result=if type(s) eq cstring then nulc else nulb;
(1≤ ∀i≤n) result=result+vs;;
return if type(s) eq cstring then repcstr(result)
           else repbstr(result);
end rep_s;

definef arith(a); /* (Utility routine.) */
if type(a) eq int or type(a) eq real then return t;
else return f;;
end arith;


definef intersect(a,b);
/* Iterate over the smaller set, checking to see if each
   member of it is also a member of the larger set. No
   type checking is done here. */
if nelt(a) le nelt(b) then smaller=a; /* reference */
                           larger=b;  /* reference */
                      else smaller=b; /* reference */
                           larger=a;; /* reference */
```

```
result=nullset;
p=nult;
x=nextelt(smaller,p);
(while x ne undef)
      if el_tf_set(x,larger) then augment_set_aok(result,x);;
      x=nextelt(smaller,p);
      end while;
return result;
end intersect;

definef slash(a,b);  /* Specifies a/b.  */
ta=type(a);  va=value(a);
tb=type(b);  vb=value(b);
if ta eq int then
      if tb eq int then return quotient(a,b);;
      if tb eq real then return repreal(va/vb);;
          /* Use BALM real div.  */
      error_type('slash(a,b), a an integer, b is neither an integer
          nor a real.'); end if ta;
if ta eq real then
      if tb eq int or tb eq real then return repreal(va/vb);;
      error_type('slash(a,b),a is real, b is neither an
           integer nor a real.'); end if ta;
if ta eq bstring then
      if tb eq bstring then return or(a,not(b));;
      error_type('slash(a,b), a is a bit string but
          b is not.'); end if ta;
```

```
error_type('slash(a,b), a is neither an integer, a real,
      nor a bit string.');
end slash;

definef quotient(a,b); /* Specifies a/b for integers.  */
va=value(a);
vb=value(b);
if vb eq 0 then error_value('quotient(a,b), b is zero.');;
/* Note: SETL uses "number-theoretic" integer division.  */
result=(abs va)/(abs vb);
if va lt 0 then result=-result-1;;
if vb lt 0 then result=-result;;
return repint(result);
end quotient;

definef dslash(a,b);  /*  Specifies a//b.  */
ta=type(a);  va=value(a);
tb=type(b);  vb=value(b);
if ta ne tb then error_type('dslash(a,b), a and b are
      of dissimilar types.');;
if ta eq int then return remainder(a,b);;
if ta eq bstring then return or(and(a,b),and(not(a),not(b)));;
if ta eq set then return symdif(a,b);
error_type('dslash(a,b),a and b are not both integers, bit
      strings, or sets.');
end dslash;

definef remainder(a,b); /*  Specifies a//b for integers.  */
/*  Notes:  (1) an error exit results if b is zero.
            (2) This is equivalent to a "modulus" function
                (the remainder is always non-negative).  */
```

```
va=value(a);
vb=value(b);
return repint(va-(vb * value(quotient(a,b))));
end remainder;

definef symdif(a,b); /* Specifies a//b for sets.  */
/*  This is coded from the relation a//b=(a u b)-(a * b),
    which probably results in a slightly faster routine than
    (a-b)u(b-a), particularly in the case where either a or b
    is small.   */
     return minus(union_args_ok(a,b), intersect(a,b));
     end symdif;

definef and(a,b);  /*  Specifies and in terms of a lower level
and which operates on equal-length boolean strings in a con-
ventional way (in particular, nulb and nulb is nulb).  */

if type(a) ne bstring or type(b) ne bstring then
      error_type('and(a,b), either a or b is not a boolean string.');;
va=value(a); vb=value(b);
if  #va ge  #vb then return repstr(va and(( #va- #vb)bin 0+vb));
else return repbstr((( #vb- #va)bin0+va) and vb);;
end and;

definef or(a,b); /*  Specifies or; see note above.  */
if type(a) ne bstring or type(b) ne bstring then
      error_type('or(a,b), either a or b is not a boolean string.');;
va=value(a); vb=value(b);
if  #va ge  #vb then return repstr(va or(( #va- #vb)bin 0+vb));
else return repbstr((( #vb- #va)bin 0+va) and vb);;
end or;
```

```
definef not(a); /* Specifies not a (and n a).  */
if type(a) ne bstring then error_type ('not(a),a is not a boolean
      string.');;
if a eq nullbstring then return nullbstring else return repbstr
      (not value(a));;
end not;

definef type(x); /*  Specifies type x.  */
if x eq undef then return undef;
else return repint(type(x));;
end type;

definef pair(x); /* Specifies pair x.  */
if type(x) eq tupl and nelt(x) eq 2 then return true;
else return false; /* Note: pair Ω  is f.  */
end pair;

definef newat; /*  Specifies newat.  */
newat_number=newat_number+1;
return repblank(newat_number);
end newat;
```

```
definef max(a,b); /*  Specifies a max b.  */
if arith(a) and arith(b) then
       if value(a) ge value(b) then return a;
       else return b;;
else error_type('max(a,b), either a or b is not a real or
       an integer.');;
end max;

definef min(a,b); /*  Specifies a min b.  */
if arith(a) and arith(b) then
       if value(a) le value(b) then return a;
       else return b;;
else error_type('min(a,b), either a or b is not a real
       or an integer.');;
end min;

definef abs(a); /*  Specifies abs a.  */
if arith(a) then
       if value(a) ge 0 then return a;
       else return pminus(a);;
else error_type('abs(a),a is neither real nor an integer.');;
end abs;

definef pminus(a); /*  Specifies -a.  */
if type(a) eq int then return repint(-value(a));;
if type(a) eq real then return repreal(-value(a));;
error_type('pminus(a), a is neither an integer nor a real.');
end pminus;

definef pplus(a); /*  Specifies +a.  */
if arith(a) then return(a);
else error_type('pplus(a), a is neither an integer nor a real.');;
end pplus;
```

```
definef floor(x); /* Specifies floor x. */
/* The SETL floor and ceiling functions convert between reals
and integers as illustrated below, on a machine whose floating
point is decimal with three significant digits.
```

x	floor x	ceiling x
3.00	3	3
3.14	3	4
-3.14	-4	-3
3	3.00	3.00
3141	3.14E3	3.15E3
-3141	-3.15E3	-3.14E3

```
On a binary machine, the results would be similar but not exactly
the same in cases such as the last two rows.  Regardless of
machine details, the following relations hold for real or integer
x.

              floor x ≤ x
              ceiling x ≥ x
              ceiling x = -floor(-x)          */

if type (x) eq real then
     r=value(x)//1.0; /* Use BALM REMAINDER. */
     if r lt 0 then r=r+1.0;;
     return repint(value(x)-r);; /* After subtracting "r",
            round to nearest integer. */
if type(x) eq int then
     /* The steps below assume the existence of a FLOAT
     function which produces exact results if the magnitude
     of the given integer is less than base exp precision for
     the machine. */
```

```
          base=2;precision=52; /* use(16,6) or (16,14) for S/360. */
          max=base exp precision;
          vx=value(x); f=1.0;
          if vx ge 0 then
               (while vx ge max) vx=vx/base; f=base * f; end while;
          else d=base-1;
               (while vx le -max)vx=(vx-d)/base;f=base * f; end while;
          end if vx;
          return repreal(f * FLOAT(vx));
     end if type;
     error_type('floor(x), x is neither a real nor an integer.');
     end floor;

     definef ceiling(a); /*  Specifies ceiling a.  */
     return pminus(floor(pminus(x)));
     end ceiling;

     definef genset(x); /*  Specifies {x1,...,xn}: x is a
     meta-tuple (use BALM list).  To invoke, code (for example)
     genset(<a,b,c>).    */
     args=x;
     result=nullset;
     (while args ne nult)
          next=hd args;
          args=tl args;
          result=augment_set(result,next);
          end while;
     return result;
     end genset;

     definef gentup(x); /*  Specifies <x1,...,xn>: x is a
     meta-tuple (use BALM list).  To invoke, code (for example)
     gentup(<a,b,c>).  */

     (∀y∈x) if y eq undef then error_type('gentup(x),a component
          of x is Ω.'); end if; end ∀y;
     h=if #x eq 0 then hash_null_tuple else hash_code(hd x);
     return <tupl,h,<#x,x>>;
     end gentup;
```

```
definef all(s,c); /* Specifies the predicate ∀x∈ s | c(x). */
/* "C" is a meta-function which returns the object-language
values "true" and "false". */
if type (s) ne set then error_type('all(s,c), s is not a set.');;
p=nult;
x=nextelt(s,p);
(while x ne undef)
     if c(x) eq false then return false;;
     x=nextelt(s,p);
     end while;
return true;   /* Used if s=nullset. */
end all;

definef any(s,c); /* Specifies ∃x ∈ s | c(x). */
return anys(x,s,c);
end any;

definef anys(x,s,c); /* Specifies ∃[x] ∈ s | c(x). */
if type (s) ne set then error_type('anys(x,s,c), s is not a set.');;
p=nult;
x=nextelt(s,p);
(while x ne undef)
     if c(x) eq true then return true;; /* (x is a parameter.) */
     x=nextelt(s,p);
     end while;
return false;   /* Note that x=undef here. */
end anys;

definef dec(a); /* Specifies dec a. */
if type(a) eq int then return external_int(a,10);;
if type(a) eq cstring then return internal_int(a,10);;
error_type('dec(a),a is neither an integer nor a cstring.');
end dec;
```

```
definef oct(a); /*  Specifies oct a.  */
if type(a) eq int then return external_int(a,8);;
if type(a) eq cstring then return internal_int(a,8);;
error_type('oct(a), a is neither an integer nor a cstring.');
end oct;

definef external_int(a,b);
/*  Converts integer "a" to a character string, using
base "b".  Note:  "b" is a meta-integer≥2 and ≤10.
Samples:  external(0,b)='0',
          external(13,10)='13',
          external(-13,8)='-15'.    */

x=value(abs(a));
if x eq 0 then return repcstr('0');;
digits='0123456789';
result=nulc;
(while x ne 0)
     result=digits((x//b)+1)+result;
     x=x/b;
     end while;
if value(a) lt 0 then result='-'+result;
return repcstr(result);
end external_int;

definef internal_int(a,b);
/*  Converts character string "a" to an integer (in internal
form), interpreting the string as being a base "b" representation
of the integer.  Note:  "b" is a meta-integer ≥2 and ≤10.
```

```
Samples:  internal('47',10) is 47,
          internal('b-b47,8) is -39,
          internal('bbb',10) is 0,
          internal(nulc,10) is 0.

There are three loops in the code below:
     1.  Locate first digit (if any), saving the sign (if any).
     2.  Calculate the result.
     3.  Check for garbage following last digit.  */

/*  Locate first digit.  */
k=1;
sign=nulc;
x=value(a);
(while i le #x doing i=i+1)
     char=x(i);
     if char eq ' ' then continue while i;;
     if char eq '-' or char eq '+' then
         if sign ne nulc then then error_value('string
            to integer conversion, string has multiple signs.');;
         sign=char;
         continue while i; end if char;
     /*  Character is neither blank, +, or -.  */
     quit;
     end while i;

/* Calculate the result.  */
map={<'0',0>,<'1',1>,<'2',2>,<'3',3>,<'4',4>,<'5',5>,<'6',6>,
     <'7',7>,<'8',8>,<'9',9>};
result=0;
(while i le #x doing i=i+1)
```

```
     if x(i) eq ' ' then i=i+1; quit;;
     digit=map(x(i));
     if digit eq Ω then error_value('string to integer
        conversion, illegal character or bad format.  Form should
        be 'bSbDb;, where b is zero or more blanks, S is +,-, or
        null, and D is zero or more digits.');;
     if digit ge base then error_value ('string to integer
        conversion, the string contains a digit equal to or
        exceeding the base.');;
     result=(b * result)+digit;
     end while i;

/*  Check for garbage (number may have been terminated by
a blank).   */
(while i le #x doing i=i+1)
     if x(i) ne ' ' then error_value ('string to integer conversion,
         a non-blank character found after last digit.');;
     end while i;
if sign eq '-' then result=-result;;
return repint(result);
end internal_int;

definef bitr(a); /*  Specifies bitr a.  */
if type(a) eq int then return bitr_int(a);;
if type(a) eq real then return bitr_int(floor(a));;
if type(a) eq bstring then return bitr_bstring(a);;
error_type('bitr(a), a is neither an integer, a real, nor
     a boolean string.');;
end bitr;

definef bitr_int(a);
/*   Converts an integer to a boolean string.
     Samples:  bitr_int(0)=nulb
               bitr_int(6)=110b.        */
```

```
x=value(a);
if x lt 0 then error value('bitr(n), n is negative.');;
result=nulb;
(while x ne 0 doing x=x/2)
     result=(if x//2 eq 1 then t else f)+result;
     end while;
return repbool(result);
end bitr_int;

definef bitr_bstring(a);
/* Converts a boolean string to a non-negative integer. */
x=value(a);
result=0; i=1;
(while i le #x doing i=i+1)
     result=(2* result)+(if x(i) eq t then 1 else 0);
     end while i;
return repint(result);
end bitr_bstring;

definef le(a,b); /*  Specifies a le b.  */
if arith(a) and arith(b) then /*  Use BALM LE.  */
     if value(a) le value(b) then return true;
     else return false;;;
if type(a) eq set and type(b) eq set then
     return le_set(a,b);;
if type(a) eq bstring and type(b) eq bstring then
     return repcstr(bitr(value(a) and not value(b)) eq 0);;
error_type('le,ge,lt,or gt; invalid data types.  Valid
     combinations are arith R arith, set R set, and
     bstring R bstring, where arith is real or int.');
end le;

definef ge(a,b); /*  Specifies a ge b.  */
return le(b,a);
end ge;
```

```
definef lt(a,b); /*  Specifies a lt b.  */
if type(a) eq bstring and type(b) eq bstring then
     return repcstr(bitr(not value(a) and value(b)) ne 0);
else return repcstr(not equal_tf(a,b) and value(le(a,b)));;
end lt;

definef gt(a,b); /*  Specifies a gt b.  */
return lt(b,a);
end gt;

definef le_set(a,b);
if nelt(a) gt nelt(b) then return false;;
p=nult;
x=nextelt(a);
(while x ne Ω doing x=nextelt(a))
     if not el_tf_set(x,b) then return false;;
     end while;
return true;
end le_set;

definef as(a,b); /*  Specifies a as b.  */
ta=type(a); va=value(a);
vb=value(b);
if ta eq vb then return a;; /*  No operation.  */
if ta eq int then
     if vb eq real then return floor(a);;
     if vb eq bstring then return bitr_int(a);;
     if vb eq cstring then return external_int(a,10);;;
if ta eq real then
     if vb eq int then return floor(a);;
     if vb eq bstring then return bitr_int(floor(a));;
     if vb eq cstring then return external_real(a,12);;
     if vb eq tupl then f=va//1.0;
                         return gentup(<repint(va-f),repreal(f)>);;;
```

```
if ta eq bstring then
     if vb eq int then return bitr_bstring(a);;
     if vb eq real then return floor(bitr_bstring(a));
     if vb eq cstring then r=nulc;(∀ b ∈ va)
                           r=r+(if b then '1' else '0');;
                           return repcstr(r);;
     if vb eq tupl then r=nult;(∀ b ∈ va)r=r+⟨b⟩;;
                        return gentup(r);;
if ta eq cstring then
     if vb eq int then return internal_int(a,10);;
     if vb eq real then return internal_real(a);;
     if vb eq bstring then r=nulb;(∀ c ∈ va)
                           if c eq '0' then b=f;
                           else if c eq '1' then b=t;
                              else error_value('character to boolean
                              string conversion, a character other
                              than "0" or "1" was found.');;;
                           r=r+b; end ∀c; return repbstr(r);;
     if vb eq tupl then r=nult; (∀c ∈ va) r=r+⟨c⟩;;
                        return gentup(r);;
if ta eq function then
     if vb eq subroutine then type(a)=subroutine;;;
     /* Note: The following is valid SETL: "s=f as subroutine:
     s(a,b);". This causes f to be invoked but its returned value
     is ignored (f is invoked for its side-effects only).  */
if ta eq set then
     if vb eq tupl then return set_as_tuple(a);;;
if ta eq tuple then
     if vb eq real then return tuple_as_real(a);;
     if vb eq bstring then return tuple_as_bstring(a);;
     if vb eq cstring then return tuple_as_cstring(a);;
     if vb eq set then return tuple_as_set(a);;;
```

```
error_type('as(a,b), invalid combination specified.  Valid
     ones are integer as integer, real, bstring or cstring;
     real as integer, real, bstring, cstring, or tuple;
     bstring as integer, real, bstring, cstring, or tuple;
     cstring as integer, real, bstring, cstring, or tuple;
     label as label; blank as blank; subroutine as
     subroutine; function as function or subroutine;
     set as set or tuple; and tuple as real, bstring, cstring,
     set, or tuple.');
end as;

definef external_real(a,ω);
```

/* Real to character string conversion. This function converts "a" to a character string of length ω or less (ω is a meta-integer), where ω must be ≥ 3. If unable to produce at least one significant digit in the result, one of the following actions occurs: (1) if $\omega < 3$, an error exit is taken, or (2) the first ω characters of the string 'OVERFLOW' are returned (if $\omega > 8$, the result string length is 8).

If at least one significant digit can be returned in valid SETL form for a real constant, then this procedure returns as many significant digits as possible within the field width ω. Insignificant zeros are eliminated if the character string representation is exact. Samples (for $\omega = 6$):

a	string	a	string
0	0.0	.0001	0.0001
1.00056	1.0006	.00012	1.2E-5
1234	1234.0	62000	6.2E4
12345	1.23E4	62001	6.20E4
-1234	-1.2E3	10^{100}	OVERFL

*/

```
/* To be coded later. */
end external_real;
```

```
definef internal_real(a);
/* Character string to real conversion.  The character string
"a" must be a valid SETL representation of a real or integer,
except that an all-blank field or a null string is valid and is
interpreted as zero, and the number may begin or end with a decimal
point.  Sample valid strings: ''(null), ' '(blanks), '0', '0.0',
'b+b0.0bEb+b020b' ("b" denotes zero or more blanks), '1E-2', '.01'.
     This algorithm breaks the string "a" up into fields and
converts each field, using "internal_int", as follows:
     1.  Scan for a period or 'E' (or end of string);
         call this point i1.
     2.  Convert characters to left of i1; call this result "k".
     3.  Scan for a blank or an 'E'; call this point i2.
     4.  Convert characters from i1 to i2; call this result "f".
     5.  Scan for an 'E'; call this point i3.
     6.  Convert characters to right of i3; call this result "e".
     7.  Result is (k+f * (10 exp(-(i2-i1-1))))*(10 exp e).  */
va=value(a);
i1=1;
(while i1 lt #va doing i1=i1+1)
     if va(i1) eq ' ' or va(i1) eq 'E' then quit;;
     end while;
k=internal_int(va(1:i1-1)); /* if L=0, then s(a:L) is nulc.  */
i2=i1;
(while i2 lt # va doing i2=i2+1)
     if va(i2) eq ' ' or va(i2) eq 'E' then quit;;
     end while;
f1=max(0,i2-i1-1);
f=internal_int(va(i1+1:f1));
i3=i2;
(while i3 lt #va doing i3=i3+1)
```

```
     if va(i3) eq 'E' then quit;;
     end while;
e=internal_int(va(i3:));
ten=repreal(10.0);
return times(plus(k,times(f,exp(ten,fl))),exp(ten,e));
end internal_real;

definef set_as_tuple(s); /*  Converts a set to a tuple. */
p=nult;
result=nult;
x=nextelt(s,p);
(while x ne undef doing x=nextelt(s,p))
     if type(x) ne set or nelt(x) ne 2 then
         error_value('set_as_tuple(s), s has a member
         that is not a set of size 2.');;
     px=nult;
     index=nextelt(x,px);
     other=nextelt(x,px);
     if type(index)ne int or index le 0 then  /* interchange */
         temp=index; index=other; other=temp; end if;
     if type(index) ne int then error_value('set_as_tuple(s),
         s has a member which does not contain a positive integer.');;
     if type(other)ne set or nelt(other) ne 1 then
         error_value('set_as_tuple(s), s has a member which does
         not contain a set of size 1.');;
     result(index)=arb_set(other);
     end while;
return gentup(result);
end set_as_tuple;

definef tuple_as_real(t);
/* Tuple to real conversion. */
if nelt(t) eq 2 then
```

```
     (i,f)=value(t);
     if type(i) eq int and type(f) eq real then
              return plus(i,f);;
else error_value('tuple to real conversion, the tuple is
     not a pair of the form ⟨integer,real⟩.');;
end tuple_as_real;

definef tuple_as_bstring(t);
/*   Tuple to boolean string conversion.  */
vt=value(t);
result=nulb;
(while vt ne nult doing vt=tl vt)
     b=hd vt;
     if type(b) ne bstring or #value(b) ne 1 then
         error_value('tuple_as_bstring(t), a component of t
         is neither t nor f.');;
     result=result+b;
     end while;
return repbstr(result);
end tuple_as_bstring;

definef tuple_as_cstring(t);
/*  Tuple to character string conversion.  */
vt=value(t);
result=nulc;
(while vt ne nult doing vt=tl vt)
     c=hd vt;
     if type(c) ne cstring or #value(c) ne 1 then
         error_value('tuple_as_bstring(t), a component of
         t is not a character string of length one.');;
     result=result+c;
     end while;
return repcstr(result);
end tuple_as_cstring;
```

```
definef tuple_as_set(t);
/* Converts a tuple to a set. */
result=nullset;
vt=value(t); index=1;
(while vt ne nult doing vt=tl vt; index=index+1;)
     h=hd vt;
     if h eq undef then continue;;
     augment_set_simply(result,genset(<repint(index),
                                     set_with_only(h)>));
     end while;
return result;
end tuple_as_set;

definef pow(s); /* Specifies pow(s)   */
if type(s) ne set then error_type('pow(s), s is
     not a set.');;
result=nullset;
(0≤∀n≤#s)
     augment_union_args_ok(result,npow(n,s));
     end ∀n;
return result;
end pow;

definef npow(n,s);  /*  Specifies npow(n,s).  */
if type(n) ne int then error_type('npow(n,s), n is not an integer.');;
if type(s) ne set then error_type('npow(n,s),s is not a set.');;
vn=value(n);
if vn lt 0 then error_value('npow(n,s), n is negative.');;
if vn gt nelt(s) then return undef;;
p=nult;
x=nextnpow(n,s,p);
result=nullset;
```

```
(while x ne undef doing x=nextnpow(n,s,p))
     augment_set_aok(result,x);
     end while;
return result;
end npow;

definef nextnpow(n,s,a);
/* Returns with "next" subset of s containing exactly n
elements.  "a" is used similarly to the "p" of nextelt; in
fact "a" contains a tuple of nextelt "addresses". This algorithm
is similar to "nexnpow" in the Notes, p. 141, but more detail
is shown and it is probably more efficient, as it does not employ
constructions of the form f[x]. Briefly, the algorithm works by
maintaining two tuples r and p, which indicate the current "state"
of the routine. The parameter "a" is the pair <r,p>.  After each
call, "r" is an n-tuple of the n members of s which are returned,
and "p" is an n-tuple of the members' "addresses" (as defined by
nextelt). On each call, the n-th component of "r" is "incremented",
if possible. If successful, the set of components of "r" is
returned. If not successful, the (n-1)-th component of "r" is
incremented, and so on, until a subset containing n distinct
members is obtained. The order of tuple "r" is as follows for
n=3 and s={a,b,c,d,e} (in nextelt-order): <a,b,c>,<a,b,d>,<a,b,e>,
<a,c,d>,<a,c,e>,<a,d,e>,<b,c,d>,...,<c,d,e>.  */
if p eq nult then /* first entry */ np=1; p(1)=nult;
            else np=n; <r,p>=a;;
(np≥ ∀j≥1)
     r(j)=nextelt(s,p(j)); /* Increment j-th component of r. */
     if r(j) eq undef then continue ∀j;;
     (j< ∀k≤n)
         p(k)=p(k-1);
         r(k)=nextelt(s,p(k));
         if r(k) eq undef then continue ∀j;;
         end ∀k;
     /* Build the set {r(i),1≤∀i≤n}.  */
     rs=nullset;
     (1≤∀i≤n) augment_set(rs,r(i)); end ∀i;
     a=⟨r,p⟩;
     return rs;
     end ∀i;
/* Through.  Even r(1) can't be incremented.  */
return undef; /* Note:  returns here immediately if s=nullset. */
end nextnpow;

definef random(x); /*  Specifies random x.  */
if type(x) eq int then return repint(random_int(x));;
if type(x) eq real then return repreal(random_real(x));;
if type(x) eq set then return random_set(x);;
error_type('random(x), x is not an integer, real, or set.');
end random;

/* Global values needed by random number generators.  */
RN_modulus=4294967296; /* 2 exp 32.  */
RN_max=RN_modulus-1;

definef RN; /*  Basic random number generator.
                Reference CACM December 1969 p.695.  */
initially seed=1;;
seed=(seed*32781)//RN_modulus;
return seed; /* Caution:RN( ) may not be very random
                in low order bits (c.f. Knuth Vol. 2, p.12).  */
end RN;

definef random_int(k);
if value(k) le 0 then error_value('random(k),k
     an integer, k is not positive.');;
```

```
return ((RN( )*value(k))/RN_modulus)+1;
     /* (Integer division) */
end random_int;

definef random_real(a);
if value(a) eq 0 then error_value('random(a), a real,
     a is zero.');;
return (RN( )*value(a))/RN_max; /* (Real division) */
end random_real;

definef random_set(s);
if s eq nullset then return undef;;
x=random_simply(s);
if not triple(x) then return x;;
/* x is the special triple for tuples in sets, that is, x
has the form <tupl,hash_code,<3,<a,Sa,0>>>.  */
y=random_set(tuple(x)(2));
tuple(x)=<head(x)>+tuple(y);
nelt(x)=1+nelt(y);
return x;
end random_set;

definef random_simply(s);
start=random_int(repint(ht_size(s)));
i=0;
(while i ne start doing i=((i+1)//ht_size(s))+1)
    if i eq 0 then i=start;;
     list=hash_table(s)(i);
     if list eq nult then continue;;
     return list(random_int(repint(#list)));
     end while;
```

```
error_impl('A set ne nullset has no used entries in its
     hash table.');
end random_simply;

definef exp(b,e); /* Specifies b exp e. */
if type(b) eq int then
     if type (e) eq int then return exp_ii(b,e);;
     if type(e) eq real then return exp_rr(floor(b),e);;;
if type(b) eq real then
     if type(e) eq int then return exp_ri(b,e);;
     if type(e) eq real then return exp_rr(b,e);;;
error_type('exp(a,b), either a or b is neither an integer nor
     a real.');
end exp;

definef exp_ii(k,n); /* k exp n for integers. */
vk=value(k); /* copy. */
vn=value(n); /* copy. */
if vn lt 0 then
     if vk eq -1 then vn=-vn; /* Note: $(-1)^{-n}=(-1)^{n}$. */
     else error_value('exp_ii(k,n), n is negative and k ne -1.');;;
/* Now vn ge 0. */
if vn eq 0 and vk eq 0 then error_value('exp(k,n),
     k and n are both 0.');;
result=1; /* initialize loop. */
(while vn ne 0 doing vn=vn/2; vk=vk * vk;)
     if (vn//2) eq 1 then result=result * vk;;
     end while;
return repint(result);
end exp_ii;
```

```
definef exp_ri(b,n); /* b exp n for real b, integer n. */
vb=value(b); /* copy */
vn=value(n); /* copy */
if vn le 0 then
     if vb eq 0 then error_value('exp_ri(b,n),b=0,n≤0.');
     else vb=1.0/vb; vn=-vn;;;/* b^-n=(1/b)^n. */
/* Now vn ge 0. */
result=1.0;
(while vn ne 0 doing vn=vn/2; vk=vk * vk;)
     if vn//2 eq 1 then result=result * vk;;
     end while;
return repreal(result);
end exp_ri;

definef exp_rr(b,e); /* b exp e for reals. */
vb=value(b);
ve=value(e);
if ve le 0 then
     if vb eq 0 then error_value('exp_rr(b,e),b=0,e≤0.');
     else vb=1.0/vb; ve=-ve;;;/* b^-e=(1/b)^e. */
/* Now ve ge 0. */
n=floor ve;
f=ve-n;
bn=exp_ri(b,repint(n));
if f eq 0 then return bn;;
/* f gt 0 */
if vb lt 0 then error_value('exp_rr(b,e),b<0,e is not
      an integer.');;
bf=EXP(vb,f);/* EXP is an undefined routine that handles
      the cases vb≥0,0<f<1. */
return repreal(value(bn) * value(bf));
end exp_rr;
```

F. Functional Application

```
definef sof(f,x); /* Specifies f{x}.  */
/* Evaluates f{x}  (for a single argument only).  */
if type(f) ne set then error_type ('f{x},f is not a set.'):;
if x eq undef then error_type('f{x},x is the undefined atom.');;
return sof_args_ok(f,x);
end sof;
```

```
definef sof_args_ok(f,x);
if f eq nullset then return nullset; end if;
index=compressed_hash_code(ht_size(f),hash_code(x));
/* Entry "index" of the hash table of f is searched for all
tuples of 2 or more components (i.e., 2 or 3) that begin with
"x". When a pair is found, the result set "s" (which is ini-
tially null) is augmented by the pair's second component. When
a triple is found, which must be of the form <x,sx,0>, s is made
equal to the union of itself and sx. */

s=nullset;
( ∀y ε hash_table(f)(index))
      if type(y) eq tupl and nelt(y) ge 2
                         and equal_tf(head(y),x) then
          y2=tuple(y)(2);
          if nelt(y) eq 2 then if y2 ne undef then
               augment_set_aok(s,y2); end if y2;
          else /* nelt(y) is 3 */ augment_union_args_ok(s,y2);
          end if nelt;
      end if type;
end ∀y;
return s;
end sof_args_ok;

definef sofn(f,x); /* Specifies f{x1,x2,...,xn}. */
/* Evaluates f{x1,x2,...,xn}. To use, the argument must
be made into a tuple (or BALM list). For example, to evaluate
f{1,2,3}, code sofn(f,<1,2,3>). */
if type(f) ne set then error_type('sofn(f,x), f is not a set.');;
if type x ne tupl or #x eq 0 then error_type
```

```
            ('sofn(f,x), x is not a tuple of length ge 1.');;
return sofn_args_ok(f,x);
end sofn;

definef sofn_args_ok(f,x);
if f eq nullset then return nullset; end if;
hx=hd x; if hx eq Ω  then go to error;;
index=compressed_hash_code(ht_size(f),hash_code(hx));
s=nullset  /*  copy  */;
/*  Entry "index" of the hash table of f is searched for all tuples
of 2 or more components (i.e., 2 or 3) that begin with "hx".  When
a pair is found, it is examined to see if it is of the form
<x1,<x2, x3,...,xn,...>> , or <x1,<x2,<x3,...,xn,...>>>,etc.
If so, the result set "s" (which is initially null) is augmented
appropriately.  When a triple is found, which must be of the form
<hx,s_hx,0>, s is augmented by all elements in S_hx, if #x=1,
or by S_hx{tail(x)}, if # x>1.  */

( ∀y ε hash_table(f)(index))
      if type(y) eq tupl and nelt(y) ge 2 and equal_tf(head(y),hx)
          then /*  Found a possible contributor to f{x}.  */
          if nelt(y) eq 2 then
              /*   Found a possible pair-contributor, e.g.
                   <a,b>,<a,<b,c,d>> , etc.   */
              yp=tuple(y)(2) /*  copy  */;
              (2≤ ∀i≤#x) /*  Note:  x is a meta-tuple.  */
                  if x(i) eq Ω  then go to error;;
                  if type(yp) ne tupl or nelt(yp) le 1 or
                       not equal_tf(head(yp),x(i)) then continue ∀y;
                  else yp=if nelt(yp) eq 2 then tuple(yp)(2)
                       else tail(yp);
                  end if type;
                  end  ∀i;
```

```
                augment_set_aok(s,yp);
            else /*  nelt(y) is 3  */
                y2=tuple(y)(2); /* y2 is the set of tails of tuples
                beginning with x1.  */
                if #x eq 1 then augment_union_args_ok(s,y2);
                else augment_union_args_ok(s,sofn(y2,tail(x)));
                end if #;
            end if nelt;
       end if type;
   end ∀y;
   return s;
error: error_type('sofn(f,x), a component of x is the undefined atom.');
   end sofn_args_ok;

   definef of(a,x); /*  Specifies a(x) for retrieval.  */
   if type(a) eq set then return of_set(a,x);;
   if type(a) eq tupl then return of_tuple(a,x);;
   if type(a) eq cstring then return of_cstring(a,x);;
   if type(a) eq bstring then return of_bstring(a,x);;
   /*  Note:  if type(a)=function, use BALM functional application.  */
   error_type('of(a,x), a is neither a set, tuple, nor string.'):
   end of;

   definef of_set(f,x);
   if x eq undef then error_type('of(set,x), x is the undefined
         atom.'); end if;
   if f eq nullset then return undef; end if;
   defined=f; /*  Initialize "multiply-defined" switch.  */
   /*  Search f for tuples beginning with x.  */
   index=compressed_hash_code(ht_size(f),hash_code(x));
```

```
( ∀y∈hash_table(f)(index))
      if type(y) eq tupl and nelt(y) ge 2 and equal_tf(head(y),x)
          then /*  Found a 2- or 3-tuple beginning with x.  */
          if defined then /*  Multiply defined.  */ return undef;
               end if;
          /*  Define the result.  */
          defined=t;
          y2=tuple(y)(2);
          if nelt(y) eq 2 then result=y2;
          else /*  nelt(y) is 3.  y2 is a set.  */
              result=if nelt(y2) eq 1 then arb_set(y2) else undef;
          end if nelt;
      end if type;
      /*  Continue scan to check for multiply-defined value.  */
      end ∀y;
return (if defined then result else undef);
end of_set;

definef of_tuple(t,i);
if type(i) ne integer then error_type('of(tuple,i), i is
      not an integer.'); end if;
if value(i) lt 1 or value(i) gt nelt(t) then return undef;;
/*  Handling of sparse tuples is ignored for the present.  */
return tuple(t)(i); /*  copy  */
end of_tuple;

define of_cstring(c,i);
if type(i) ne integer then error_type('of(cstring,i'); i is
      not an integer.'); end if;
vi=value(i); vc=value(c);
```

```
if vi lt 1 or vi gt #vc then return undef; end if;
return repcstr(vc(vi));
end of_cstring;

definef of_bstring(b,i);
if type(i) ne integer then error_type('of(bstring,i), i  is
      not an integer.'); end if;
vi=value(i); vb=value(b);
if vi lt 1 or vi gt #vb then return undef; end if;
return repbstr(vb(vi));
end of_bstring;

definef ofn(f,x); /*  Specifies f(x1,x2,...,xn).  */
/*  To use this function, the argument x must be made into
a tuple (or BALM list).  For example, to evaluate f(1,2), code
ofn(f,<1,2>).  */
if type(f) ne set then error_type('ofn(f,x), f is not a set.');;
if type x ne tupl or #x eq 0 then error_type
      ('ofn(f,x), x is not a tuple of length ge 1.');;
return ofn_args_ok(f,x);
end ofn;

definef ofn_args_ok(f,x);
if f eq nullset then return nullset; end if;
hx=hd x; /*  Note: x is a meta-tuple.  */
if hx eq Ω    then go to error;;
index=compressed_hash_code(ht_size(f), hash_code(hx));
defined=f; /*  Initialize "multiply defined" switch.  */
(∀y ε hash_table(f)(index))
      if type(y) eq tupl and nelt(y) ge 2 and equal_tf(head(y),hx)
          then /*  Found a possible result for f(x).  */
          if nelt(y) eq 2 then
```

```
                    /* Found a possible pair that will define f(x).  */
                    yp=tuple(y)(2) /*  copy  */;
                    (2≤ ∀i≤ #x)
                       if x(i) eq Ω  then go to error;;
                       if type(yp) ne tupl or nelt(yp) le 1 or
                            not equal_tf(head(yp),x(i)) then continue ∀y;
                       else yp=if nelt(yp) eq 2 then tuple(yp)(2)
                            else tail(yp);
                       end if type;
                       end ∀i;
                    /* yp is a result of f(x).  */
                    if defined then  /* Multiply defined   */
                       return undef; end if;
                    defined=t;
                    result=yp;
                 else /*  nelt(y) is 3.  */
                    y2=tuple(y)(2); /*  y2 is the set of tails of tuples
                    beginning with x1.  */
                    if #x eq 1 then /*   y2 contains the results.  */
                       if nelt(y2) eq 1 and not defined then
                           defined=t; result=y2; else return undef; end if;
                    else /*   #x gt 1.  Continue search.  */
                       result=ofn_args_ok(y2,tail(x));
                       if result ne undef then if not defined then
                           defined=t;else return undef;end if;end if result;
                    end if #x;
                    end if nelt;
             end if type;
       end ∀y;
       return(if defined then result else undef);
error: error_type('ofn(f,x), a component of x is the undefined atom.');
       end ofn_args_ok;
```

```
          definef nextelt(s,p);
          /*  Used in ∀,∃, set former, etc., this routine provides
          the next member in set "s".  The second parameter, "p", is a
          kind of "address" of the member being returned.  On first call,
          the caller should have p=nult.  Between calls, neither s nor p
          may be altered by the caller.  When there are no more members
          in s, a value of "undef" is returned.  In addition, p is set
          to Ω , for error-checking purposes.
                The address p of a member x is defined (recursively) as a
          meta-triple <index,list,pp>, where index is the position that x
          occupies in the hash table, list is (a reference to) the list
          hash-table(s)(index), starting with component y, and pp is Ω
          if y is not a triple.  If  y is a triple, it must be of the form
          <x(1),s,0>, and in this case pp is the address, in s, of the
          remaining components of x.
                The overall structure of the algorithm below is to advance
          pp, and, if that cannot be done, to advance in the list, and,
          if that cannot be done, to advance the index.   */

          if p eq Ω  then error_impl('nextelt(s,p),p is Ω .  Forgot to
                initialize p to nult or omitted check for end of loop.');
                end if;
          /*  Restore where we left off on last call.  */
          <index, list,pp>=if p eq nult then <0,nult,Ω>   else p;

next_tup: if pp ne Ω  then  /* working on a triple   */
                <first,set,->=tuple(hd list);
                x=nextelt(set,pp);
                if x eq undef then go to next_in_list; end if;
                p=<index,list,pp>;
                return<tupl,hash_code(first),<1+nelt(x),
                                                <first>+tuple(x)>> ;
```

```
               end if pp;
         /*  pp is Ω  (usual case).  Either previous item is not a
         triple or it is but we are through with it.  */
next_in_list:if #list ge 2 then
               list=tl list;
               go to build;
               end if  #;

         /*  Advance index.  */
         (while index lt ht_size(s))
               index=index+1;
               list=hash_table(s)(index);
               if list ne nult then go to build;
               end while index;
         /*  No more members in set s, or s is null.   */
         p= Ω ;
         return undef;

    build: item=hd list;
         if triple (item) then pp=nult; go to next_tup; end if;
         p=<index,list,Ω>;
         return item;
         end nextelt;

         definef bof(f,s); /*  Specifies f[s].  */
         if type(f) ne set then
               error_type('bof(f,s),f is not a set.'); end if;
         if type(s) eq set then return bof_set(f,s);;
         if type(s) eq tupl then return bof_tuple(f,s);;
         if type(s) eq cstring then return bof_cstring(f,s);;
if type(s) eq bstring then return bof_bstring(f,s);;
error_type('bof(f,s), s is not a set, tuple, or string.');
end bof;

definef bof_set(f,s); /*  Coded from the definition:
      f[s]=[+: ∀x ε s]f{x}.  */
if s eq nullset then return undef;;
p=nult;
result=nullset;
next=nextelt(s,p);
(while next ne undef)
      augment_union_args_ok(result,sof_args_ok(f,next));
      next=nextelt(s,p);
      end while;
return result;
end bof_set;

definef bof_tuple(f,t); /*   Coded from the definition:
      f[t]=[+: ∀x ε t]<f(x)>,
      and if there exists a defined x ε  t such that f(x)= Ω   ,
      then an error exit results.    */
result=nult; /*   Meta-tuple(use BALM list).   */
count=0;
todo=t; /*   copy   */
(while todo ne nulltupl)
      x=head(todo);
      todo=tail(todo);
      if x eq undef then continue; /*   For sparse tuples.   */
      y=of_set(f,x);
      if y eq undef then error_type('bof(f,t), t is a tuple,
            f(t(i)) is undefined for some t(i) ne Ω  .');;
      result=result+<value(y)>; /*   BALM APPEND may be used.   */
```

```
                count=count+1;
                end while todo;
          if count eq 0 then return undef;
          return <tupl,h,<count,result>>;
          end bof_tuple;

          definef bof_cstring(f,s); /*   Coded from the definition:
                f[s]=[+: ∀c ε s]f(c), and if
                there exists a c ε s such that f(x) is undefined, then
                an error exit results.  Note that the result, f(s(1))+...
                +f(s(n)), may be an integer, bit string, character string,
                set, or tuple (anything for which + is a valid operator). */

          if s eq nullcstring then return undef;;
          result=of_set(f,repcstr(value(s)(1)));
          if result eq undef then go to error;;
          (2≤ ∀i≤ #value(s))
                y=of_set(f,repcstr(value(s)(i)));
                if y eq undef then go to error;
                result=plus(result,y);
                end ∀i;
          return result;

error:    error_type('bof(f,s), there exists a character c in s for which
                f(c) is not defined.');
          end bof_cstring;

          definef bof_bstring(f,s); /*  Coded from the definition;
                see remarks for bof_cstring.  This algorithm is similar
                to that used in bof_cstring, except that f(x) is only
                evaluated twice, once for x=0b and once for x=1b, for
```

```
              improved efficiency.  (It is assumed that bit string "s"
              is most likely to consist of mixed zeros and ones.)    */

        if s eq nullbstring then return undef;;
        f0=of_set(f,repbstr(0b));
        f1=of_set(f,repbstr(1b));
        result=if value(s)(1) eq 0b then f0 else f1;
        if result eq undef then go to error;;
        (2≤ ∀i≤ #value(s))
              y=if value(s)(i) eq 0b then f0 else f1;
              if y eq undef then go to error;;
              result=plus(result,y);
              end ∀i;
        return result;

error:  error_type('bof(f,s), there exists a bit b in s for which f(b)
              is not defined.');
              end bof_bstring;

        definef bofn(f,x); /*   Specifies f[x1,...,xn].  x must be
              a meta-tuple whose components are SETL sets.  */
        if type(f) ne set then error_type('bofn(f,x),x is not a set.');;
        if type x ne tupl or #x eq 0 then error_type
              ('bofn(f,x),x is not a tuple of length ge 1.');;
        return bofn_args_ok(f,x);
        end bofn;

        definef bofn_args_ok(f,x); /*  Coded from the relations
              f[x,y]=(f[x][y],etc.  Requires  #x evaluations of
              bof_set, and hence  #x1+...+ #xn evaluations of
              nextelt, union, and f{x}.  Evaluation stops if an
              intermediate result is null, and the remaining arguments
              are not checked.   */
```

```
result=f; /*   copy   */
remaining=x; /*   copy   */
(while remaining ne nult and result ne nullset)
      next=hd remaining;
      if type(next) ne set then error_type('bofn(f,x),
            a component of x is not a set.');;
      result=bof_set(result,next);
      remaining=tl remaining;
      end while;
return result; end bofn_args_ok;
```

V. Index

V. <u>Index</u> (concluded)