SETL Newsletter No. 108 May 1973

APL-SETL, an extension of SETL achieved from user varied semantics.

G. Jennings

Table of Contents

1. Introduction

We describe an extension of SETL, APL-SETL which contains primitives similar to those of Iverson's APL. We use the mechanisms in SETL Newsletter 76 for the specification of the semantics of subroutine and function invocation as a function of the types of the arguments. APL-SETL differs from Iverson's APL language in minor ways. The from and kind declarations defined and discussed in SETL NL 76 are our vehicles for defining APL-SETL.

APL provides four classes of primitive operators -- monadic and dyadic functions on scalars and arrays, reduction, inner and outer product, and primitive mixed functions. We will treat each of these classes of operators in turn giving for each a block of declarations, i.e. *kind* and *from* statements. We will also give code in SETL for effecting the semantics of some of the primitives in each group. We put aside the issue of right-to-left execution order which we feel to be of relatively small consequence. The left to right parse of SETL will be assumed.

APL provides scalars and arrays as data types. The scalars may take on as values either real numbers or one of the boolean primitives, t and f. The components of an array must be either real numbers, boolean primitives, or characters. In addition, the components of an array must be of a single type. We remark at this point that the real number '3.14' is not implemented in the same way as is the array whose only component is '3.14', although in many APL primitives the latter may be used instead of a scalar.

Real numbers will be the usual SETL objects as will be the boolean primitives t and f. These objects will be said to be of kind *real* and *bool*. Arrays will be either of kind *aplarray*, *boolarray*, and *chararray* if their components are real numbers, boolean primitives or characters respectively. Variables belonging to one of these classes will be generically termed *arrays*, whereas scalars whether of kind *real* or *bool* will be said to be scalars. The arrays with no components will be called *empvector*. We do not specify that *empvector* belongs to any of the classes *aplarray*, *boolarray*, or *chararray*.

An array of kind *aplarray*, *boolarray* and *chararray* is indexed by a finite number of indices which may vary dynamically throughout the execution of a program. Hence the number of indices and the range of each must be retained throughout the execution of a program. We choose to represent an object of kind *array* as a SETL pair. The first component of each pair is a tuple which determines the number of indices and the range of each index. The second component of an object of kind *aplarray* is a tuple whose components are the elements of the array. For example, the array

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \end{pmatrix}$$

where s_{ij} is the component in the i-th row and j-th column would be implemented as

$$<<2,3>, <s_{11},s_{12},s_{13},s_{21},s_{22},s_{23}>>$$

The tuple <2,3> indicates that the first index takes on the values 1 or 2 and that the second index varies from 1 to 3. In the following we will assume that 1 is the origin of all indices. That is, the least value of each index is one, not zero. This restriction simplifies the exposition. Note that the components of each array are ordered in row major order, i.e. the last index varies most rapidly.

Similarly, objects of kind *boolarray* are implemented as a pair of the form

<dimvector,bstring>

where *dimvector* is a tuple of the form of the first component of an object of kind *aplarray*. *bstring* is a SETL bit string, i.e. a string of boolean t's and f's, and is said to be of kind *bitstring*. Items of kind *chararray* are implemented similarly. However the second component is a character string, which is of kind *charstring*.

We first give a block of declarations which contains this structural information.

```
from tℓ aplarray get tuplescal;
from tuplescal(integer) get real;
from tℓ boolarray get bitstring;
from bitstring(integer) get bool;
from tℓ chararray get charstring;
from hd array get dimvector;
from dimvector(integer) get integer;
```

2. Monadic and Dyadic Arithmetic Primitives

We begin by discussing our implementation of the monadic and dyadic arithmetic operations. APL allows expressions of the form

numberpassed ← numberpassed + (grade > 70) .

The boolean valued expression (grade > 70) is treated as the real number 1 if the value of the boolean expression is true and zero, otherwise. Our choice for the representation of objects of kind *bool* and *boolarray* as pairs containing SETL bit strings is an efficient representation of bit strings. This representation forces conversion of objects of kind *bool* to objects of kind *real* prior to evaluation of an expression of the form above. This conversion could be avoided if objects of kind *bool* and *boolarray* were implemented alternatively as a restricted class of objects of kind *real* and *aplarray* respectively. That is, the real number 1 would represent t and the real number 0 would represent f. We choose the economical representation of objects of kind *boolarray* and pay the expense of making conversions as required.

In APL source text an expression of the form

... a+b ...

may occur at a time when each of *a* and *b* is either a *scalar* or an *array*. If one argument is a scalar, that scalar is treated as an array each of whose components is that scalar of the same dimensions as the other argument and the calculation is performed by adding the components pairwise. An array which contains a single element is treated as a scalar for the purposes of these conformability considerations.

The APL primitive *reduction* will produce a scalar result, if the operands are vectors, i.e. indexed by a single index, or an array if the operand has more than one index. Other primitive mixed functions produce a result the number of whose indices is different from that of the argument. In particular, *compression* produces an array, the number of whose dimensions depends on one of the arguments.

Rather than require the user of APL-SETL to correctly distinguish between scalars and arrays, we dynamically determine whether an argument is an array or a scalar. For example, prior to evaluating

... a+b ...

a and *b* must be examined to determine if the number and range of the indices of each array are compatible.

On the other hand, it is not difficult for the reader to specify if a variable is of kind *real*, *bool*, or *char*, because each APL primitive produces unambiquously a result

of one of these three classes. Namescopes may be used as logical parentheses which will allow the type of the value of a token to change from *real* to *bool*. Hence, we require it to be possible for expressions of the form

real + bool

to be distinguised in the source code from expressions of the form

... bool + real ...

or ... bool + bool ...

We outline declarations with all of these combinations of kinds of arguments. The compilation process will include code for making conversions where required.

We now give a **complete set of declarations** which provide the semantics of the monadic and dyadic infix operators applied to two scalars each of which is of kind *real*. We have used symbols ⍟ and ? as infix operators. To realize APL-SETL in an environment which does not recognize these as infix operators, the reader should substitute tokens like <u>star</u> and <u>question</u>.

```
from + real get real using scalplus;
 "   -  "    "    "    "   scalnegative;
 "   ×  "    "    "    "   scalsignum;
 "   ÷  "    "    "    "   scalreciprocal;
 "   ⌈  "    "    "    "   scalceiling;
 "   ⌊  "    "    "    "   scalfloor;
 "   *  "    "    "    "   scalexponential;
 "   ⍟  "    "    "    "   scalnatlog;
 "   |  "    "    "    "   scalmagnitude;
 "   !  "    "    "    "   scalfactorial;
 "   ?  "    "    "    "   scalroll;
 "   ∘  "    "    "    "   scalpitimes;
from ∼ bool get bool using scalnot;

from real + real get real using scaladd;
 "    "   -  "    "    "    "   scalminus;
 "    "   *  "    "    "    "   scaltimes;
 "    "   /  "    "    "    "   scaldivide;
 "    "   ⌈  "    "    "    "   scalmaximum;
 "    "   ⌊  "    "    "    "   scalminimum;
 "    "   *  "    "    "    "   scalpower;
 "    "   ⍟  "    "    "    "   scallogarithm;
 "    "   |  "    "    "    "   scalresidue;
```

```
from real ! real get aplscalar using scalbinomial;
 "     "  ∘  "    "      "        "   scalcircular;
 "     "  <  "    "  bool         "   scallt;
 "     "  ≤  "    "      "        "   scalle;
 "     "  =  "    "      "        "   scaleq;
 "     "  ≥  "    "      "        "   scalge;
 "     "  >  "    "      "        "   scalgt;
 "     "  ≠  "    "      "        "   scalne;
 "   bool ∧ bool  "      "        "   scaland;
 "     "  ∨  "    "      "        "   scalor;
 "     "  ⍲  "    "      "        "   scalnand;
 "     "  ⍱  "    "      "        "   scalnor;
```

Most of the routines required by the declarations made above are available by modification of FORTRAN library routines.

These operations on scalars are extended in APL to operations on arrays. An expression of the form

... a+b ...

is admissible when *a* and *b* are arrays or scalars only if *a* and *b* are conformable. If *a* and *b* are both arrays, the number of indices and the range of each index must be the same for each of *a* and *b*. The exception to this is the case that one of the arguments is an array which contains a single component. In this case, this argument is treated as though it were a scalar. One of the arguments, for example *a*, can be a scalar in which case, this argument is handled as though it were an array of the same shape as *b* each of whose components is equal to the scalar. It is possible that each of *a* and *b* is a scalar. Testing for conformability is done dynamically in each of the routines mentioned below.

For the cases that one or more arguments is of type *bool* additional declarations are required. These can be produced mechanically from the statements we gave above. We now give examples of these declarations.

```
from real + bool get real using scaladdfarg;
 "   bool + real  "    "     "  scaladdsarg;
 "   bool + bool  "  bool    "  scaladdboth;
```

and

```
from boolarray + aplarray  get  aplarray using addfarg;
 "    aplarray + boolarray  "  boolarray   "   addsarg;
 "   boolarray +     "      "      "       "   addboth;
```

We now indicate the code required to implement the dyadic arithmetic primitives in arrays. The arguments of each primitive must be conformable arrays. Two arrays are conformable if their dimension vectors are identically the same, both are scalars, or one of the arguments is a scalar or an array with a single component. In the latter case, the argument is treated as if it were an array of the same shape as the argument which is equal to a constant in each component. The case of two scalars may occur and is handled by an invocation of the primitive designed for the scalar case. If the arguments are not conformable, *empvector*, the array with no components, is returned.

We now give the routine which determines if two arrays are conformable. If only one argument is a scalar conformable returns implicitly an array of the correct dimensions each of whose components is that scalar.

```
definef conformable(arg1,arg2);
/* atype determines if argument is 'scalar' or 'array' */
initially where = {<t,t,botharr>, <f,t,fscal>,
                   <t,f,secscal>,<f,f,bothscal>};;
firstargarray = atype(arg1) eq 'array';
secargarray = atype(arg2) eq 'array';
go to where(<firstargarray,secargarray>);
botharr: if(hd arg1) eq (hd arg2)
         then return <'botharr', tl arg1, tl arg2, hd arg1>;
         end if;
return if hd arg1 eq <1>
     then <'firstargscal', (tl arg1)(1), tl arg2, hd arg2>;
     else if hd arg2 eq <1>
     then <'secondargscal', tl arg1, (tl arg1)(1), hd arg1>;
     else <'notconformable'>;
fscal:  return <'firstargscal', arg1, tl arg2, hd arg1>;
secscal: return <'secscal', tl arg2, arg2, hd arg2>;
bothscal: return <'bothscal', arg1, arg2>;
     end conformable;
```

Four functions are required to implement the addition of two arrays, that is, *add*, *addfarg*, *addsarg*, and *addboth*.

We give a macro for generating all four of these functions.

```
  macro dextend(arrfn, mod, scalfn, conv1, conv2);
  definef arrfn → mod(a,b);
  initially where = {<'notconformable',notconform>,
       <'firstargscal',firstscal>,<'secondaryscal',secscal>,
       <'bothscal',bothscal>,<'botharr',botharr>};;
  resconform = conformable(a,b); go to where(hd resconform);
  notconform: return empvector;
  bothscal:   <-,a,b> = resconform;
              return scalfn (a,b);
  firstscal:  <-,a,b,dimres> = resconform;
       valresult = [+: 1 ≤ j ≤ #b]<scalfn (conv1(a),conv2(b(j)))>;
       return <dimres,valresult>;
  secscal:    <-,a,b,dimres> = resconform;
       valresult = [+: 1 ≤ j ≤ #a]<scalfn (conv1(a(j)), conv2(b))>;
       return <dimres,valresult>;
       botharr; <-,a,b,dimres> = resconform;
       valresult = [+: 1 ≤ j ≤ #a]<scalfn (convI(a(j)), conv2(b(j)))>;
       return <dimres,valresult>;
 end add;
endm;
```

The functions *conv1* and *conv2* will perform conversion of scalars of type *bool* to scalars of type *real* as required by the types of the arguments.

We avoid, in this style, the construction of an array whose components are the same scalar and then the subsequent extraction of those components.

The reader will recognize that expansion of the following macro will generate code for *add*, *addfarg*, *addsarg*, and *addboth*. *Convert* is the global set {<t,1.0>, <f, 0.0>}.

```
macro extendall(scalfn,arrfn);
     dextend(arrfn,,scalfn,,);
     dextend(arrfn, farg, scalfn, convert,);
     dextend(arrfn, sarg, scalfn,, convert);
     dextend(arrfn, both, scalfn, convert, convert);
endm;
```

We give the expansion for *add*.

```
definef add(a,b);
initially where = {<'notconformable',notconform>,
      <'firstargscal',firstscal>,<'secondaryscal',secscal>,
      <'bothscal',bothscal>,<'botharr',botharr>};;
resconform = conformable(a,b); go to where(hd resconform);
notconform: return empvector;
            <-,a,b> = resconform;
bothscal:   return scaladd(a,b);
firstscal:  <-,a,b,dimres> = resconform;
     valresult = [+: 1 ≤ j ≤ #b]<scaladd(a,b(j))>;
     return <dimres,valresult>;
secscal:    <-,a,b,dimres> = resconform;
     valresult = [+: 1 ≤ j ≤ #a]<scaladd(a(j),b)>;
     return <dimres,valresult>;
     botharr; <-,a,b,dimres> = resconform;
       valresult = [+: 1 ≤ j ≤ #a]<scaladd(a(j),b(j))>;
       return <dimres,valresult>;
end add;
```

Similarly the remaining functions which we require can be obtained by expanding this macro with different values for the arguments. We give a partial listing of such functions.

```
extendall(add,scaladd);
extendall(minus,scalminus);
            .
            .
            .
extendall(nequal,scalnequal);
```

The remaining lines required should be clear to the reader.

For the algorithms given in SETL NL 76 to interpret correctly a construction like

a + b

where **a** is a scalar of type *real* and b is an array of type *aplarray*, we include reversion stipulations

```
revert real(aplarray);
revert bool(boolarray);
```

3. <u>Reduction, inner product, and outer product</u>.

We now discuss the implementation of the reduction, inner product, and outer product primitives, The syntax in APL of the reduction primitive is f|v where f is a system supplied dyadic function like +, *, or ∧ ; and v is a vector. The result is the scalar v(1) <u>f</u> f(2) <u>f</u> ... . An array indexed by more than one index may be reduced along the i-th coordinate by a construction of the form f/[i]a. The result is an array of one fewer dimensions. If no coordinate is specified, reduction is made along the last coordinate. If *v* or *a* is a scalar at run time, the result is the identity element for the function which is the other argument. Our construction will allow a user supplied routine to be used in the reduction primitive in addition to system supplied primitives.

We consider only those functions whose arguments are of kind *real* and whose result is of kind *real*. The modifications required to permit objects of kind *bool* to be arguments of real functions and to include functions whose result is of kind *bool* are mechanical in nature and should be clear from the remarks above.

We use

fn <u>*slash*</u> v

as the form of the reduction primitive in APL-SETL which replaces the APL syntax

fn/v

The function *fn* must be a dyadic operator. To reduce along the i-th coordinate, we use

fn <u>*slash*</u> <i,a>

This replaces the APL construction

fn/[i]a

We now give declarations to support the reduction primitive in APL-SETL.

```
 from   <integer, aplarray> get indexaplobj;
 fromf  realfn slash aplarray get aplarray using dreduction;
 fromf  realfn slash indexaplobj get aplarray using reduction;
```

Realfn is a kind which designates the class of functions for which the arguments and result are of kind *real*.

We suggest at this time a variant of the *from* declarations discussed in SETL NL 76. The statements

```
          from a op b get typec  using newop
          from f(a,b) get typed  using newfn
```

cause the code fragments

```
         <call, fneval, newop, a, b, t1>

         <call, fneval, newfn, a, b, t2>
```

to replace

```
         <call, fneval, op, a, b, t1>
                                                  (1)
         <call, fneval, f, a, b, t2>          .
```

The routine *fneval* determines if *f* or *op* is a set or a coded function as a preliminary to determining the code sequence to execute to interpret each code fragment.

We propose a class of declarations of the form

```
          fromf a op b get typec using newop
          fromf f(a,b) get typed using newfn
```

which cause the code fragments

```
         <call, newop, op, a, b, t1>

         <call, newfn, f, a, b, t2>
```

to replace (1). The result of this new class of declarations is that the function (*op* or *f*) becomes an argument to the routine mentioned in the using clause rather than being replaced. The modification of the other from clauses should be clear as is the requirement to modify the algorithms in section 7 of SETL NL 76.

With this vehicle for user specification of the semantics of functions and subroutines, we include the *inner product* primitive which appears in APL as

af.gb

where *f* and *g* are system supplied primitives.

We mime this in APL-SETL as

a f *dot* g b

where f and g are any dyadic infix operators, and *dot* is a distinguished token. In APL-SETL this will be passed as

(a f *dot*) g b

To support this we give the declarations

```
fromf aplarray realfn dot get realpair using makpair;
fromf realpair realfn aplarray get aplarray using innerprod;
```

will cause *a* f *dot* to be evaluated by *makpair*(f,a,dot) which returns the SETL pair <a,f>. The routine *innerprod* will then have arguments *g*, *<a,f>* and *b* on which the algorithm for inner product can be implemented.

The APL-SETL syntax for outer product is

a circle dot f b

This mimes the APL construction

$a_{\circ}.fb$

The declarations required are

```
fromf a circle dot get aplobj using getleftarg;
fromf aplobj realfn aplarray get aplarray using outerprod;
```

Objects of kind *aplobj* are represented in the same manner as objects of kind *aplarray* but are distinguished so that the second declaration will cause the invocation of *outerprod*.

We do not give code for *innerprod*, *outerprod*, or *reduction*. The semantic effect should be clear. Techniques for encoding these routines can be found in the next section on primitive mixed functions.

4. Primitive Mixed Functions

a. Declarations

We now discuss the implementation of the primitive mixed functions. Page 3.38, IBM publication APL/360-OS and APL/360-DOS Users Manual (GH 20-0906-0) contains a complete list of these primitives. In the declarations below we indicate which arguments are *vectors*, which are arrays indexed by a single index, by the tokens *aplvector* and *boolvector*. These tokens are synonyms for *aplarray* and *boolarray* respectively. We will use the distinction in our discussion at the end.

We give declarations for all of the primitive mixed functions with the exception of the *index* primitive which is discussed in the next section on sinister constructs. We give declarations when the arguments are of kind *aplarray* and *boolarray*. The extensions to *chararray* are left to the reader. As above, we use the APL designators for these infix operators. Suitable tokens should be substituted when the system is used in an environment which does not support these designators as infix operators.

```
                 alias aplvector, aplarray;  alias boolvector boolarray;
           size: from ρ array get aplvector using size;
        reshape: from aplarray ρ aplarray get aplarray using reshape;
                 from aplarray ρ boolarray get aplarray using reshape;
          ravel: from , aplarray get aplvector using ravel;
       catenate: from  aplarray,aplarray get aplarray using catenate;
                 from  boolarray,boolarray get boolarray using catenate;
indexgenerator:  from ι real get aplarray using indexgen;
        indexof: from aplvector ι aplarray get aplvector using indexof;
                 from boolvector ι boolarray get aplvector using indexof;
           take: from aplvector ↑ aplarray get aplarray using take;
                 from aplvector ↑ boolarray get boolarray using take;
```

Name	Sign[1]	Definition or example[2]
Size	ρA	ρP ↔ 4 ρE ↔ 3 4 ρ5 ↔ ι0
Reshape	VρA	Reshape A to dimension V 3 4ρι12 ↔ E 12ρE ↔ ι12 0ρE ↔ ι0
Ravel	,A	,A ↔ (×/ρA)ρA ,E ↔ι12 ρ,5 ↔ 1
Catenate	V,V	P,ι2 ↔ 2 3 5 7 1 2 'T','HIS' ↔ 'THIS'
Index[3] [4]	V[A] M[A;A] A[A;.. ..;A]	P[2] ↔3 P[4 3 2 1] ↔7 5 3 2 E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ ABCD EFGH IJKL
Index generator[3]	ιS	First S integers ι4 ↔ 1 2 3 4 ι0 ↔ an empty vector
Index of[3]	VιA	Least index of A in V, or 1+ρV Pι3 ↔2 PιE ↔ 5 1 2 5 3 5 4 5 5 5 5 5 4 4ι4 ↔ 1
Take	V↑A	Take (drop) |V[I] first elements on coordinate I. (Last if V[I]<0) 2 3↑X ↔ ABC EFG
Drop	V↓A	¯2↑P ↔ 5 7
Grade up[5]	⍋A	The permutation which would order A (ascending or descending) ⍋3 5 3 2 ↔ 4 1 3 2
Grade down[5]	⍒A	⍒3 5 3 2 ↔ 2 1 3 4
Compress[5]	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 0 1/[1]E ↔ 1 2 3 4 9 10 11 12 ↔ 1 0 1⌿E
Expand[5]	V\A	1 0 1\ι2 ↔ 1 0 2 1 0 1 1 1\X ↔ A BCD E FGH I JKL
Reverse[5]	⌽A	⌽X ↔ DCBA HGFE LKJI ⌽[1]X ↔ ⊖X ↔ IJKL EFGH ABCD ⌽P ↔ 7 5 3 2
Rotate[5]	A⌽A	3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P 1 0 ¯1⌽X ↔ BCDA EFGH LIJK
Transpose	V⍉A	Coordinate I of A becomes coordinate V[I] of result 2 1⍉X ↔ AEI BFJ CGK DHL 1 1⍉E ↔ 1 6 11
	⍉A	Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∊A	ρW∊Y ↔ ρW P∊ι4 ↔ 1 1 0 0 E∊P ↔ 0 1 1 0 1 0 1 0 0 0 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal[3]	S?S	W?Y ↔ Random deal of W elements from ιY

Table 3.8: PRIMITIVE MIXED FUNCTIONS

```
       drop: from aplvector ↓ aplarray get aplarray using drop;
             from aplvector ↓ boolarray get boolarray using drop;

    gradeup: from ⍋ aplarray get aplvector using gradeup;
             from ⍋ boolarray get aplvector using gradeup;

  gradedown: from ⍒ aplarray  get aplvector using gradedown;
             from ⍒ boolarray get aplvector using gradedown;

   compress: from aplvector / aplarray  get aplarray using compressconv;
             from boolvector/ aplarray  get aplarray using compress;

             from aplvector / boolarray get boolarray using compressconv;
             from boolvector/ boolarray get boolarray using compress;

     expand: from aplvector / aplarray  get aplarray using expandconv;
             from boolvector/ aplarray  get aplarray using expand;

             from aplvector / boolarray get boolarray using expandconv;
             from boolvector/ boolarray get boolarray using expand;

    reverse: from ⌽ aplarray  get aplarray  using reverse;
             from ⌽ boolarray get boolarray using reverse;

     rotate: from aplvector ⌽ aplarray  get aplarray  using rotate;
             from aplvector ⌽ boolarray get boolarray using rotate;

  transpose: from aplarray  ⍉ aplarray  get aplarray  using transpose;
             from aplarray  ⍉ boolarray get boolarray using transpose;
             from           ⍉ aplarray  get boolarray using transposed;
             from           ⍉ boolarray get boolarray using transposed;

 membership: from aplarray  ∈ aplarray get boolarray  using memberof;

     decode: from aplarray    ⊤   aplarray get real using decode;

     encode: from aplarray    ⊥   aplarray get aplvector using encode;
```

We give code in SETL for all of these primitives but for *reverse*, *rotate*, *transpose*, *deal*, and *membership*. The techniques in the code for the primitives that we implement can be used to implement these other primitives.

b. Implementation

We first give code **for a routine** *lookup* which extracts a component of an array, i.e. which calculates $a(i_1,i_2,\ldots,i_k)$ where k is the rank of *a*, that is the number of indices of the array *a*. Next, we give the sinister version of this routine. In *lookup* each of $i_1,i_2,\ldots,i_k$ must lie between 1 and the maximum index of *a* in that component. Violation of this constraint in the dexter format results in Ω being returned. In a sinister construction, a noop results.

The function *linearloc(a,dimvector)* calculates the position in <u>tℓ</u> a of the component whose indices are the components of *dimvector*. We provide this as a separate routine to isolate this feature which is dependent upon the implementation of objects of kind *array*. We remark we have chosen a representation of arrays in which the last index varies most rapidly.

```
definef lookup(a,dimvector);

(load)

return if linearloc(a,dimvector) is where eq Ω

     then Ω else (tℓ a)(where);

end load;

(store t)

if linearloc(a,dimvector) is where ne Ω

     then (tℓ a)(where) = t;;

return;
end store;
end lookup;


definef linearloc(a,dimvector);

dima = hd a;

/* check that dima and dimvector are conformable */

if not((#dima eq #dimvector) and

     [and: 1 ≤ j ≤ #dima](1 le dimvector(j) and dimvector(j)

                                                      le dima(j)))

then return Ω;
endif;
/* calculate location in  tℓ a of elements with indices
                                  dimvector */
where = dimvector(1) - 1;
(1 < ∀j ≤ #dima)

where = where * dima(j) + dimvector(j)-1;

end ∀j;

return (where+1);

end linearloc;
```

In the following discussion, we use the kindtype *vector* to designate an array of kind *real*, *bool*, or *char* which is indexed by a single coordinate. Similarly, we use the kindtype *matrix* to designate an *array* referenced by two indices. We now give macros for checking the types of arguments of the SETL implementation of APL primitives.

We give one macro for checking that a variable is a vector. The action upon detection of an error is to return *empvector*.

```
macro cvector(arg);
/* check that arg is a vector -- the action upon the
     detection of an error is to return empvector */
if not (#arg eq 2 and #(hd arg) eq 1)
     then return empvector;
endif;
endm;
```

We now give a similar macro in which the second parameter is a SETL code fragment which specifies the action to be taken upon the detection of an error.

```
macro cvector2(arg,altaction);
/* check that arg is a vector - altaction is a SETL code fragment
   which specifies the action to be taken on the detection of an
                                                          error*/
if not (#arg eq 2 and #(hd arg) eq 1)
  then altaction;
endif;
endm;
```

We also give a macro which determines if an argument is a *scalar* of kind, ***real***, or is an array with one component. In the latter case the scalar is extracted and made available for subsequent processing.

```
macro cscalar(news,olds);
/* news is the value of the scalar contained in  olds*/
news = if #olds eq 2 and #olds(2) eq 1
     then (tℓ olds)(1) else olds;
if type news ne real  then return empvector;;
endm;
```

We give a macro to determine if a variable is an array. We do not check the type of each component.

```
macro carray(arg);
if #arg eq 2 and type arg(1) eq tuple  and
        type arg(2) eq tuple then continue;

     else return empvector;
endm;
```

Now we give code for the *decode* primitive which produces a single object of kind *real* from an *aplvector* whose components are the representation of thatscalar in the number system designated by the second argument. The second argument may be either a vector or a scalar. In the latter case, the number system is powers of that scalar. Note that an array with a single component is treated as a scalar.

```
definef decode(vector,radix);
kind radix(aplarray);  kind vector(aplarray);

cvector(vector);
cvector2(radix,go to scalar); /*if radix not a vector
                                then go to scalar */
if #radix(2) eq 1 then radix = radix(2,1)  go to scalar;;

valr = radix(2); valv = vector(2);
if #valr gt (#valv+1) then return empvector;;
/* conformability error */
result = valr(#valv);
(#valr > ∀j ≥ 1)  result = result * valv(j) + valr(j);;


return result;
/* number system is    powers of a scalar */
scalar: if not(type(radix) eq real)then return empvector;;


rad = radix;   kind rad(real);  result(real);
result = valr(#valr);
```

```
(#valr > ∀j ≥ 1) result=result*rad+valr(j);;
return result;
end decode;
```

As *rad* and *result* have been declared to be of kind *real*, and *valv* is of kind *aplarray*, the multiplication and addition in the line

```
            result = result + rad * valv(j)
```

are compiled into invocations of *scalprod* and *scaladd*. We now give code for *encode* which is an inverse to *decade*.

```
definef encode(v,sarg);
/* check that sarg designates a scalar and v designates a vector*/
cscalar(s,sarg);  cvector2(v, go to scalar);
result = nult;  first=1;  tlv = tℓ v;  product = tlv(1);
while (product lt s) first=first+1; product=product*tlv(j);;
  result = nult;
(first ≥ ∀j > 1)
  product = product/tlv(j)
  result = result+<floor(s/product) is s>;
end ∀j;
return <<#result>, result>;
scalar: cscalar(tlv,v); /* tlv is a scalar */
result = nult;
while(s ne 0)
result = <s - floor(s/tlv) *tlv is s> + result;;
return <<#result, result>>;
```

We now give routines for the *size*, *reshape*, *ravel*, and *catenate* primitives. The semantics of these primitives should be clear from the code.

```
definef size(a);
carray(a); /* check that a is an array */
    return <<#(hd a)>, hd a>;
end size;
```

We remark that the object returned is of kind *aplarray* and is a *vector*. Next we give the *reshape* function.

```
definef reshape(v,a);
carray(a); cvector(v);
tla = tℓ a;
lengtha = #(tℓ a);
/* convert components to integers */
lengthres = convscalint([*: 1 ≤ j ≤ #tℓ v] (tℓ v)(j));
nrcopiesa  = lengthres/lengtha;
nrcompfrag = lengthres - nrcopiesa * lengtha;
result = nult;
(1 ≤ ∀j ≤ nrcopiesa) result=result+tℓ a;;



result = result + (tℓ a)(1: nrcompfrag);
return <tℓ v, result>;
end reshape;
```

The *ravel* primitive is straightforward in the implementation that we have chosen. The result is always of kind *vector*.

```
definef ravel(a);
carray(a);
return <<#tℓ a>, tℓ a>;
end ravel;
```

The last of this group is the *catenate* primitive.

```
definef catenate(v1,v2);
cvector(v1);  cvector(v2);
return <<hd hd v1 + hd hd v2>, v1(2) + v2(2)>;
end catenate;
```

We now code the primitive *indexgenerator* which generates an *aplvector* whose components are the integers from 1 to its argument.

```
definef indexgen(olds);
cscalar(s,olds);
i = convscalint(s);
/* converts scalar to SETL integer */
if i eq 0 then return empvector;;
return <<i>, [*: 1 ≤ k ≤ i]<convintscal(k)>>;
end indexgen;
/* convintscal(·) changes a SETL integer to  SETL real     */
```

Now we turn to the *indexof* function which calculates the least index in a vector v of each component of an array a. The result is an array of the same shape as a.

```
definef indexof(vector,a);
carray(a); cvector(vector);
tlv = tℓ vector;
tla = tℓ a;
result = nult;
(1 ≤ ∀j ≤ #tla)
result = result + <if(1 ≤ ∃k ≤ #tlv|tlv(k) eq tla(j))
     then k else #(tlv)+1>;
end ∀j;
return <hd a, convintscal[result]>;
end indexof;
```

The next primitive we implement is *take* which is written in APL as V ↑ A. The primitive *take* produces an array whose components are some of those of A. If V[I] is positive the first V[I] components of the Ith index of A are chosen. If V[I] is negative the last V[I] componets of the I-th component of A are chosen.

```
definef take(vector,a);
cvector(vector); carray(a);
/* components of tlv are of kind  real  - convert to integers*/
tlv = convscalint[tℓ v];
minarg = nult; maxarg = nult;
/* calculate minimum and maximum arguments */
(1 ≤ ∀j ≤ #tlv)
if (tlv(j) is newi lt 0)
   then minarg = minarg + <-newi+1>; maxarg=maxarg+<hda(j)>;
   else minarg = minarg + <1>;   maxarg = maxarg + <newi>;

endif;
  end ∀j;
/* calculate dimension vector for result */
dimres = [+: 1 ≤ j ≤ #minarg]<maxarg(j) - minarg(j)>;
<hda, tla> = a;
/* iterate over all coordinates of a */
indicesa = [*: 1 ≤ j ≤ #hda]<1>;
k = 1; result = nult;
(while indicesa(1) le hda(1)
     doing indicesa = augment(indicesa, hda);  k=k+1;)

if [and: 1 ≤ j ≤ #hda](minarg(j) le indicesa(j) and
                    indicesa(j) le maxarg(j))
  then result = result + <tla(k)>;
endif;
end while;
return <dimres, result>;
end take;
```

The function *augment* increments the components of *indesca* using the dimension information contained in *hda*. Recall that the last index varies most rapidly in our implementation.

```
definef augment(indices,dimvector);
j = #dimvector;
indices(j) = indices(j)+1;
(while indices(j) gt dimvector(j) and j ge 1 doing j=j-1;)
indices(j) = 1;
indices(j-1)=indices(j-1)+1;
end while;
return indices;
end augment;
```

The drop primitive is implemented in terms of the take primitive. The routine *negative(,)* return the *aplarray* whose components are those of the argument with the sign changed.

```
definef drop(vector,a   );
return take(negative(vector),a   );
end drop;
```

The *gradeup* and *gradedown* primitives stand in the same relationship to each other as do the *take* and *drop* primitives. The code for these primitives follows.

```
definef gradeup(vector);
tlv = tℓ vector;
set = {<tlv(j),j>, 1 ≤ j ≤ #tlv};
sorttuple = sort(set);
/* sort is a sorting algorithm which orders the first
   components of elements of set */
return <<#set>, tℓ[sorttuple]>;
end gradeup;
```

```
/* tℓ[v] is the tuple <tℓ v(1), tℓ v(2), ...>  */
definef gradedown(vector);
return gradeup(negative(vector));
end gradedown;
```

We now turn to the compression primitive U/X. U should be a vector of kind *boolarray* or of kind *bool* in which case t/X equals X and f/X is *empvector*. Constructions of the form U/[I]A and U/A in APL result in compression occurring along the Ith coordinate in the former case and along the last coordinate in the case that no coordinate is specified. No declarations were made for these cases. The compression primitive eliminates those components of X or A for which the corresponding entry U is f and retains the remainder.

```
definef compress(u,indexarray)
<i,a> = indexarray;
if u eq t then return a;;
if u eq f then return empvector;;
cvector(u);  <-,tlu> = u;  <hda,tla> = a;
if i gt #hda or hda(i) ne #tlu
   then return empvector;;
/* iterate over all components of  a */
indicesa = [*: 1 ≤ j ≤ #hda]<1>;
k = 1; result = nult;
```

```
 (while indicesa(1) le hda(1)
      doing indicesa = augment(indicesa,hda); k = k+1;)
 if tlu(convscalint(hda(i)))
   then result = result + <tla(k)>;
 endif;
 end while;
 dimres = hda;  dimres(i) = [+: 1 ≤ j ≤ #tlu|tlu(j)] 1 ;
 return <dimres, result>;
 end compress;
```

The expansion primitive is similar, but where in the compression primitive components are elided, zeroes, f's, or blanks are inserted into the result when the vector u contains an f. The appearance of t in place of U esults in the original array being returned. We now give code for this primitive.

```
definef expand(u,indexa);
initially fillchar = {<'bool',f>,<'real', aplo>;
                      <'char',' '>};;
<i,a> = indexa;
cvector(u); carray(a);
<hda,tla> = a; <hdu,tlu> = u;
fill = fillchar(type tla(1));
/* fill is inserted if components of u  are f */
nrtruth = [+: 1 ≤ j ≤ #tlu|tlu(j)] 1 ;
if nrtruth ne hda(1)
  then return empvector;
endif;
```

```
dimres = hda;  dimres(i) = #tlu;

result = nult;

/* iterate over allcomponents of result */

indexres = [+: 1 ≤ j ≤ #dimres]<1>;

cntra = 1;

(while indexres(1) le dimres(1)

     doing indexres = augment(indexres,dimres);

if tlu(indexres(i))

     then result = result + <tla(cntra)>;

          cntra  = cntra + 1;

     else result = result + <fill>;

endif;

end while;

return <dimres,result>;

end expand;
```

5. Assignment, the index primitive, and sinister constructions.

We first give a discussion of the *index* primitive in a general dexter format. We then discuss the assignment primitive as a restricted inverse to the index primitive. Finally, we remark that a class of primitive operations makes sense in a sinister construction. We indicate how to include these constructs in APL-SETL.

The *index* primitive changes the domain of an array. For example, V[A], when V is a vector and the components of A are integers (i.e. of kind *real* but with values in the integers), is an array whose indices are those of A and whose range is that of the vector V. V[A] at any tuple of indices $i_1, i_2, \ldots, i_k$ is calculated as $V[A[i_1; i_2; \ldots; i_k]]$. The *index* primitive generalizes to an array A_0 with an arbitrary number of indices. For example, $A_0[A_1; \ldots; A_k]$, where k is the number of indices in A_0 is a function from the cross product of the domains of $A_1, A_2, \ldots, A_k$ into the range of A_0. In APL the omission of an array in any coordinate of A_0 defaults to the identity vector with a length equal to the maximum index of A in that coordinate. In APL-SETL, this construction will be mimed by

$$A_0[\text{tuple}]$$

where *tuple* is a SETL tuple whose components are of kind *aplarray*. Elision of an index will be marked by the appearance of an asterisk rather than by unseparated commas.

If a component of *tuple* is an integer, i.e. a scalar of kind *real*, then the number of coordinates of the result is reduced by one for each scalar that appears in *tuple*. If all of the components of *tuple* are *integers* then the result is the scalar at those indices in A_0.

So long as all of the components of *tuple* are integers or arrays which are permutations on a domain which is a subset of the range of an index of A_0, then

$$A_0\text{[tuple]} = obj;$$

is well defined. The components of A_0 indicated by *tuple* must be changed so that a subsequent evaluation of A_0[tuple] will yield *obj*. We now give declarations to support the *index* primitive in dexter and sinister format.

```
from <aplarray,aplarray,-> get apltuple;
from aplarray[apltuple] get aplarray using index;
from boolarray[apltuple] get boolarray using index;
forℓ aplarray[apltuple] = aplarray use sindex;
forℓ boolarray[apltuple]= boolarray use sindex;
```

The routine *sindex* must include a (dynamic) evaluation of the range and number of indices of the right-hand side which must agree with the range and number indicated by the left-hand side. In the sinister construction, the components of *apltuple* are restricted to be real scalars or *vectors* of kind aplarray. As we have remarked above, determination of this from the source code alone or even with the assistance of declarations supplied by the programmer is not possible without detailed knowledge of the value of variables as different stages of the exectuion of the program.

We now remark that a natural interpretation may be given to the *transpose*, *reverse* and *rotate* primitives in a sinister construction. We now give declarations.

```
forℓ ϕ aplarray = aplarray use sinreverse;
forℓ ϕ boolarray= boolarray use sinreverse;
forℓ aplvector ∅ aplarray = aplarray use sintranspose;
forℓ aplvector ∅ boolarray = aplvector use sintranspose;
forℓ ∅ aplarray = aplarray use sinreverse;
forℓ ∅ boolarray= boolarray use sinreverse;
```

We omit code for *sinreverse*, *sintranspose*, and *sinreverse*.