SISAL 1.2: A Brief Introduction and Tutorial

David C. Cann
Computing Research Group, L-306
Lawrence Livermore National Laboratory.
P.O. Box 808, Livermore, CA 94550
cann@sisal.llnl.gov

Abstract

The intent of this tutorial is to introduce the SISAL 1.2 programming language and highlight all its features (except for error values). For simplicity, discussion is kept brief, but the number of examples and illustrations is kept high. In addition to code excerpts, we present seventeen complete SISAL programs ranging from Factorial to Gaussian Elimination and the simulation of Gel Chromatography. We also provide an overview of the FIBRE input and output language.

1 Introduction

This tutorial illustrates and exemplifies SISAL 1.2, a functional language for parallel numerical computation [6]. Digital Equipment Corporation, the University of Manchester (England), Lawrence Livermore National Laboratory, and Colorado State University began the SISAL language project as a cooperative venture in 1983. The primary goal evolved to become the achievement of sequential and parallel execution performance superior to programs written in conventional languages [2]. SISAL programs cannot contain side effects or aliases; hence, they are concise, easy to understand, and amenable to automatic parallelization. A SISAL computation does not alter memory; it defines semantically immutable values. Regardless of the target architecture, SISAL programs are determinate.

The intent of this tutorial is to introduce the SISAL programming language and highlight all its features (except for error values). For simplicity and illustration, we provide numerous source code excerpts, both contrived and real. To begin, we introduce the basic organization and structure of a SISAL program, and then consider all the constructs and features. To complete the document, we provide a full listing of seventeen real SISAL programs ranging from Factorial to Gaussian Elimination and the simulation of Gel Chromatography [4]. For several of these programs we present both the sequential and parallel implementations. For the Chromatography code, we list the FORTRAN from which we wrote the SISAL. We also give an overview of the FIBRE language [7]. FIBRE specifies the format for the textual input and output of SISAL programs.

This document is an outgrowth of the Sisal Scientific Computing Initiative, started at Lawrence Livermore National Laboratory to educate scientists in the art of functional computing, and to transfer the technology to industry [1,8].

2 Basic Program Organization

A SISAL program is a collection of functions:

```
define main

function Inc( a:integer returns integer )
  a+1
end function

function main( returns integer )
  Inc(2009)         % this expression defines 2010
end function
```

The functions may occupy separate source files, return multiple values (be of multiple arity), and recurse. They cannot reference outside values other than their arguments, however.

For convenience, SISAL source is free-form and case-insensitive. Further, definition before use is strictly enforced and data-typing is strict. However, explicit typing is required only for function headers; empty array and stream definitions; and union definitions. Type equivalence is structural.

Source commentary, as illustrated above, begins with a percent sign and ends with a newline character. Space, tab, and newline characters are "white space" in SISAL source, as are comments. White space must appear between adjacent constants, names, or reserved words to prevent misinterpretation.

With regard to name scoping, the rules are similar to those of PASCAL and C. That is, a name is visible throughout its scope of definition and any enclosing scope unless blocked by a redefinition. Multiple definitions within the same scope, however, are not allowed (SISAL is a single assignment language). A "scope" in SISAL is a source file, function definition, let-in block, or loop body, as we shall see shortly.

2.1 Export Declarations

A SISAL source file must identify those locally defined functions that have global visibility. At least one function must be so identified, and the specification must appear at the top of the file in a `define` clause. For example, function `main` in the above example is exported, but function `Inc` is not. In the following example, functions in other files may reference `Sqr` and `Cube`:

```
% this declaration must appear at the top of the file
define Sqr,Cube

function Sqr( x:real returns real )
  exp(x,2.0)        % exp is a Sisal intrinsic, here defining x**2.0
end function

function Cube( x:real returns real )
```

```
    Sqr(x)*x
  end function
```

Most SISAL compilers will recognize a specific exported function as the program's entry point. The Optimizing SISAL Compiler (OSC), for example, recognizes function `main` [3].

2.2 Type Definitions

Following the export declarations, one can establish one or more type-names to serve as syntactic shorthands in function headers, etc. A type-name can appear anywhere a type specification may appear within a source file. For example,

```
  define concat

  function concat( s1:array[character]; s2:array[character]
                   returns array[character] )
    s1||s2             % define the concatenation of s1 and s2
  end function
```

and

```
  define concat

  type string = array[ character ];  % a syntactic type abstraction

  function concat( s1:string; s2:string returns string )
    s1||s2
  end function
```

are identical programs, each returning the concatenation of its two input arguments (which are arrays of characters). The second is cleaner and easier to maintain.

We provide a more complex example of type-name definition below. We discuss the semantics of the various types in later sections of this tutorial.

```
  % Data_Record represents a record having 10 fields.
  type Data_Record = record [
                       MaxStep : integer;
                       StopTime : double_real;
                       IntPlot, MaxPlots, NXCell, NYCell : integer;
                       LStrip, LDebug, LNewton : boolean;
                       GodOrder : integer ];

  % OneDim represents an array of double reals
  type OneDim = array[ double_real ];

  % PlaneArray represents an array of OneDims
```

```
type PlaneArray = array[ OneDim ];

% SystemArray represents an array of PlaneArrays
type SystemArray = array[ PlaneArray ];

% SystemStreamType represents a stream of SystemArrays
type SystemStreamType = stream[ SystemArray ];

% Plot represents a record having 7 fields
type Plot = record [ NX, NY : integer;
                     VMax, VMin : double_real;
                     XCellCenters, YCellCenters : array[ double_real ];
                     UOut : PlaneArray ];

% PlotStreamType represents a stream of Plots
type PlotStreamType = stream[ Plot ];
```

2.3 Import Declarations

Immediately following the optional type definitions, one can provide one or more import declarations, each giving the type signature of a single imported function. An imported function is one that can be referenced within the file but is defined elsewhere; that is, the function is defined and listed in the export declaration of another SISAL source file, or is a library routine. Several examples follow:

```
global acos( num:double_real returns double_real )
global sqrt( num:double_real returns double_real )
global sin(  num:double_real returns double_real )
global cos(  num:double_real returns double_real )

global Csub( cnum1,cnum2:CplexReal returns CplexReal )

global U_V_Spectral( mx,jx,jxx:integer; epsi:ArrReal1;
                     p,c:ArrCplexReal
                     returns  ArrCplexReal, ArrCplexReal )
```

Note that unlike type definitions, semicolons do not separate import declarations. The use or disuse of semicolons in SISAL is construct specific. In general, they serve as separators. Further, the formal argument names, although required, are superfluous in this context.

2.4 Forward Declarations

Following the optional imports, one can provide one or more forward declarations. Their primary use is in recursion where the name, arguments, and result types of a function must be made available before the function's actual definition. Remember that SISAL requires a strict adherence to definition before use. An example follows:

```
% provide an advanced signature for Goo
forward function Goo( x:real returns real )

function Foo( C:real returns real )
  % calls Goo in second branch of the conditional
  if ( C > 1000.0 ) then 1.0 else Goo(C+1.0) end if
end function

% now define Goo which indirectly calls itself via function Foo
function Goo( V:real returns real )
  Foo(V)
end function
```

The forward declaration allows `Goo` to be reference within function `Foo`, which is defined before `Goo`. Note that the argument names in the two headers for `Goo` do not match.

Like import declarations, semicolons do not separate forward declarations.

2.5 Function Definitions

The remainder of the source file is reserved for actual function definitions. Such a definition comprises a function signature and one or more expressions defining the results. The following function defines the approximate area under `x*x+1.0` between `Start` and `Finish`:

```
function area( start,finish:real; gran:integer returns real )
  let
    width := (finish-start)/real(gran);
  in
    for I in 1, gran
      x := start+((real(I-1))*width);
      y := (x*x)+1.0;
      v := y*width;
    returns value of sum v
    end for
  end let
end function
```

For convenience, a function may return two or more values of possibly different type. The following function computes the area and circumference of a circle with radius `r`:

```
function circle( r:real returns real, real )
  % area       circumference
  3.14159*r*r, 3.14159*r*2.0
end function
```

Embedded function definitions are possible as they are in PASCAL. For example, consider the following recoding of function `area`:

```
function area( start,finish:real; gran:integer returns real )

  function f( x:real returns real )       % The embedded function
    (x*x)+1.0
  end function

  let
    width := (finish-start)/real(gran);
  in
    for I in 1, gran
      x := start+((real(I-1))*width);
      y := f(x);                          % call the embedded function
      v := y*width;
    returns value of sum v
    end for
  end let
end function
```

The embedded function `f` is only visible to the expressions composing function `area`. As expected, embedded definitions must proceed the expressions of the enclosing function.

2.6 An Organizational Summary

To summarize the basic organization of a SISAL source file, we present an example that uses each of the constructs introduced above. The program applies a general filter to a sequence of double real values.

```
% Main is the only exported function
define Main

% illustrates type convenience
type double = double_real;
type OneDim = array[ double ];

%
% Function Main applies a general filter A of M elements to a sequence
% X of N elements.
%
%    Y[i] = SUMMATION(j=1,M) of A[i] * X[i+j-1]     for i = 1,2,...N-M+1
%
% Inputs: M, Cycles   where N = M * Cycles
%

% illustrates importation of a library routine
global sqrt( x:double returns double )

% Init will be used before its definition
```

```
forward function Init( W,X:integer returns OneDim, array[double_real] )

function Main( M,Cycles:integer; returns OneDim )

  % illustrates an embedded function
  function Conv( M,N:integer; A,X:OneDim; returns OneDim )
    for I in 1, N-M+1
      V := for J in 1, M
           returns value of sum A[J]*X[I+J-1]
           end for;
    returns array of V
    end for
  end function

  let
    N   := M*Cycles;
    A,X := inIT(M,N);   % illustrates use before definition
  in
    Conv( M,N, A,X )
  end let
end function

% illustrates multiple arity and case insensitivity: InIt vs inIT and
% Init. This function is of arity 2, returning two arrays.
function InIt( M,N:integer returns OneDim, OneDim )
  for I in 1, M returns array of sqrt(double_real(I)) end for,
  for I in 1, N returns array of sqrt(double_real(I)) end for
end function
```

3 Scalar Types, Operations, and Expressions

In this section, we enumerate and illustrate the scalar expressions of SISAL. Overall, SISAL's expression syntax and precedence rules are identical to those of FORTRAN, PASCAL, and C.

As discussed in the previous section, SISAL functions comprise one or more side effect free expressions. These expressions read and define values from SISAL's value domain, which is subdivided into disjoint subdomains that form SISAL's data types. There are the familiar basic types found in other high level languages (boolean, character, integer, real, and double_real) as well as the standard compound types (arrays, records, and unions). The stream data type, also part of SISAL, is special in that it is not found in most other high level languages for numerical computation. Implicit type lifting as well as complex and function data types are not supported in SISAL 1.2.

Each scalar data type has its own domain of values, and is characterized by a set of operations that create and transform values in that domain. All the standard relational operators are supported (`=`, `~=`, `>`, `<`, `<=`, and `>=`). They can be applied to any of the scalar types. The result is always a boolean. For character data, relational comparisons are made using numeric representations from the ASCII character set. The boolean constants are `true` and `false`. An example use of each relational operator follows:

```
p[k,l] = 0.0D0          % equality
LastPlotNum ~= NStep    % inequality
shockdir > 0            % greater than
theta < tes[nt]         % less than
itno <= niter           % less than or equal
rho >= res[nr+1]        % greater than or equal
```

The standard boolean operations are supported (`&`, `|`, `~`). They take and define booleans:

```
(j = jx) & (m = mx)     % boolean and
lp = 1 | mod(lp,2) ~= 0  % boolean or
~(j = 1 & m = 1)        % boolean not
```

As one might expect, SISAL supports the standard arithmetic operators (`+`, `-`, `*`, `/`, `mod`, `exp`, `abs`, `max`, `min`). Result types must match argument types, which can be of integer, real, or double real type (except for `mod` which only works with integer data). Argument types must match as well. Examples follow:

```
ske + eint               % addition
gamma - 1.0D0            % subtraction
energy[k,l] * mass[k,l]  % multiplication
kappa/dtnph              % division
-s2                      % negation
mod(l,640)               % modulus: the remainder of l/640
exp( temp[k,l], 2.0D0 )  % exponentiation: temp[k,l]**2.0D0
abs(tu)                  % absolute value
max( tempcaV, 0.0001D0 ) % maximum value
min( dtc, dte )          % minimum value
```

Because SISAL does not support implicit type conversion, a variety of explicit conversion operations are supported. Like `mod` and `exp`, they appear as predefined functions:

```
floor(-3.67E0)      % yields integer -4
integer(23.9189D2)  % yields integer 2392: floor(23.9289D2+0.5D0)
trunc(-2.8)         % yields integer -2
real(1)             % yields real 1.0E0
double_real(2)      % yields double_real 2.0D0
integer('A')        % yields integer 65 (from the ASCII sequence)
character(65)       % yields character 'A' (from the ASCII sequence)
```

4 Let Expressions

The purpose of the `let` expression is syntactic convenience, allowing the association of value names with subexpressions. The names can be used freely within their scope of definition as shorthand substitutes for the subexpressions they denote.

The following function defines the mean and standard deviation of its four arguments. The arity of the enclosed `let` expression is two:

```
function stats( a,b,c,d:real returns real, real )

  let
    % for convenience, name the subexpressions

    mean := (a+b+c+d)/4.0;          % here semicolons act as separators
                                    % and are required
    a_diff := (a-mean)*(a-mean);
    b_diff := (b-mean)*(b-mean);
    c_diff := (c-mean)*(c-mean);
    d_diff := (d-mean)*(d-mean);

    sum_diff := a_diff + b_diff + c_diff + d_diff; % this semicolon is
                                                   % not required
  in
    % expressions defining the let's result, which is of arity two.
    mean, sqrt( sum_diff/3.0 )
  end let

  % note that mean, a_diff..d_diff, and sum_diff are NOT visible outside
  % the let, and definition before use is mandatory.

end function
```

A `let` expression that defines a single arithmetic value may itself appear within an arithmetic expression. This is true for all SISAL constructs.

```
% define the area of the circle having radius r
Area := 3.14159 * let
                    % note that the result arity does not inhibit the
                    % number of value names that can be defined here
                    rsqr := r*r
                  in
                    % this let has single arity; hence, the let can
                    % be within an arithmetic expression!
                    rsqr
                  end let;
```

A value name can only represent one subexpression throughout its scope of definition; hence the following code fragment is illegal SISAL:

```
let
  A := 3.1456;
  A := r*A;        % this redefinition is ILLEGAL Sisal!
```

```
in
  A*r
end let
```

An inner scope, however, can reintroduce a value name defined in an outer scope. The following recoding of the area expression is legal SISAL:

```
% define the area of the circle having radius r
let
  At1 := 3.14159;
  At2 := let
           At2 := let At1 := r*r in At1 end let
         in
           At2
         end let * At1;
in
  At2
end if
```

5 Conditionals

SISAL supports an `if` construct for conditional definitions. Like the `let`, an `if` that defines a single arithmetic value may appear within an arithmetic expression. The syntax is conventional and familiar:

```
if ( x1[l] < 0.0D0 ) then 0.0D0 else qmult end if
```

The expression following the `if` is the test expression; it must define a boolean. The expression lists following the `then` and `else` reserved words give the `true` and `false` arms. The arms must define the same number of values having the same corresponding types. The results are those defined by the first arm if the expression defines `true`, else those defined by the second arm if the expression defines `false`.

For convenience, compound conditionals are supported. Consider

```
ib := if (x[l] >= xright) then 1
      else (x[l] < xleft) then 1
      elseif (y[l] < 0.0D0) then 1
      elseif (y[l] >= ymax) then 1
      else 0
      end if;
```

which is equivalent to

```
ib := if (x[l] >= xright) then 1
      else
```

```
        if (x[l] < xleft) then 1
        else
          if (y[l] < 0.0D0) then 1
          else
            if (y[l] >= ymax) then 1
            else 0
            end if
          end if
        end if
      end if;
```

Regardless, the boolean expressions are evaluated in a top-down lexicographical order, stopping at the first that defines `true`. The following illustrates a conditional of multiple arity:

```
% conditional having arity 2
xm1,xb1 := if (civ[1].itype ~= 2) then
             let
               del1 := cv[1].rr - cv[2].rr
             in
               (cv[1].zz - cv[2].zz)/del1,
               (cv[2].zz*cv[1].rr - cv[1].zz*cv[2].rr)/del1
             end let
           else
             0.0d0, 0.0d0
           end if;
```

The next two conditionals represent illegal SISAL. The first is not structurally symmetric and the second is not type symmetric:

```
% the branches have different arities: 2 and 1
if ( A[i] = 4 ) then 100, 200 else 300 end if,

% the branches define values of differing type: integer and real
if ( A[i+1] = 4 ) then 100 else 300.0 end if
```

6 Arrays

A SISAL array is a one-dimensional collection of homogeneous values indexed by simple integer expressions. Several simple array type definitions follow:

```
type OneI = array[ integer ];      % array of integers
type OneR = array[ real ];         % array of reals
type OneD = array[ double_real ];  % array of double_reals
type OneB = array[ boolean ];      % array of booleans
type OneC = array[ character ];    % array of characters
```

Multidimensional arrays are possible by declaring array constituents; that is by defining vectors of vectors:

```
type TwoI = array[ OneI ];          % array of arrays of integers
type TwoR = array[ OneR ];          % array of arrays of reals
type TwoD = array[ OneD ];          % array of arrays of double_reals
type TwoB = array[ OneB ];          % array of arrays of booleans
type TwoC = array[ OneC ];          % array of arrays of characters
```

Like all other SISAL compound types, array type conformity is structural; hence, the following five definitions are identical to the previous five:

```
type T_TwoI = array[ array[ integer ] ];      % similar to TwoI
type T_TwoR = array[ array[ real ] ];         % similar to TwoR
type T_TwoD = array[ array[ double_real ] ];  % similar to TwoD
type T_TwoB = array[ array[ boolean ] ];      % similar to TwoB
type T_TwoC = array[ array[ character ];      % similar to TwoC
```

Note that size and bounds information is not part of an array declaration. SISAL arrays are dynamic. Their size and bounds are defined at run time.

6.1 Simple Array Definitions

SISAL supports four canonical intrinsics for defining arrays. The first two are via the `array of` and `value of catenate` operations discussed in Section 10. The third is via the array generator. An example follows:

```
% array[double_real] having lower bound 2, upper bound 15, and size 14.

f := array[2: 0.144005d0, 0.177286d0, 0.212769d0, 0.249946d0, 0.288322d0,
              0.327420d0, 0.366798d0, 0.406054d0, 0.444830d0, 0.482815d0,
              0.997333d0, 0.997677d0, 0.997978d0, 0.998242d0 ];
```

An array generator can define an empty array as well:

```
e1 := array OneI [];            % see the definition of OneI above
e2 := array array[integer] [];  % e1 and e2 are equivalent
```

Here the lower bound is always one and the type signature is mandatory. The following generators together define an array of arrays:

```
t := let
       t1 := array[1:  79.0d0,    99.0d0,   125.0d0,   157.0d0,
                      198.0d0,   250.0d0,   315.0d0,   397.0d0,
                      500.0d0,   630.0d0,   974.0d0,  1000.0d0,
                     1260.0d0,  1587.0d0,  2000.0d0,  2520.0d0,
```

```
                   3175.0d0,  4000.0d0,  5040.0d0,   6350.0d0,
                   8000.0d0, 10079.0d0, 12699.0d0,  16000.0d0,
                  20159.0d0, 25398.0d0, 32000.0d0,  40317.0d0,
                  50797.0d0, 64000.0d0, 80635.0d0, 101594.0d0 ];


    t2 := array[1:  20.0d0,    25.0d0,    31.0d0,    39.0d0,
                    50.0d0,    63.0d0,    79.0d0,    99.0d0,
                   125.0d0,   157.0d0,   198.0d0,   250.0d0,
                   315.0d0,   397.0d0,   500.0d0,   630.0d0,
                   974.0d0,  1000.0d0,  1260.0d0,  1587.0d0,
                  2000.0d0,  2520.0d0,  3175.0d0,  4000.0d0,
                  5040.0d0,  6350.0d0,  8000.0d0, 10079.0d0,
                 12699.0d0, 16000.0d0, 20159.0d0, 25398.0d0 ];
  in
    array [1: t1, t2]   % array[array[double_real]]
  end let;
```

The fourth possible generator is `array_fill`. For example,

```
z := array_fill(1,100,0.0e0);  % array[real]
```

yields a 100 element array of zeros. In this example, the defined lower bound is one (defined by the first argument). The following expression builds an empty array of booleans having a lower bound of three:

```
e3 := array_fill(3,2,true);  % array[boolean]
```

An array whose upper bound is less than its lower bound is defined to be empty.

Syntactic convenience is supported for character array definitions. For example, `message1` and `message2`, shown below, represent equivalent expressions:

```
message1 := "hello world!!!";   % the lower bound is 1

message2 := array [1: 'h', 'e', 'l', 'l', 'o', ' ',
                      'w', 'o', 'r', 'l', 'd', '!', '!', '!' ];
```

6.2 Component Selection

The syntax of simple array component selection is conventional. For example, using the previous definitions of `f`, `t`, and `z` above,

```
f[3]
```

yields `0.177286d0`,

```
t[1,8]   % shorthand notation for t[1][8], which is also legal Sisal
```

yields `397.0d0`,

```
t[1]
```

yields

```
array[1:  79.0d0,    99.0d0,   125.0d0,   157.0d0,
         198.0d0,   250.0d0,   315.0d0,   397.0d0,
         500.0d0,   630.0d0,   974.0d0,  1000.0d0,
        1260.0d0,  1587.0d0,  2000.0d0,  2520.0d0,
        3175.0d0,  4000.0d0,  5040.0d0,   6350.0d0,
        8000.0d0, 10079.0d0, 12699.0d0,  16000.0d0,
       20159.0d0, 25398.0d0, 32000.0d0,  40317.0d0,
       50797.0d0, 64000.0d0, 80635.0d0, 101594.0d0 ];
```

and `z[99]` yields zero.

Section 10 illustrates a convenient method for scattering array components across product form loop bodies.

6.3 Size and Bound Queries

SISAL supports three intrinsics for obtaining the characteristics of an array. Each takes a single array as input. The array component type is irrelevant. The first intrinsic, `array_size`, yields the size of the argument. An empty array has size zero. The second, `array_liml`, yields the lower bound of the argument. The third, `array_limh`, yields the upper bound. For example, the following expression defines `6`, `-3`, and `2` respectively:

```
let
  ExampleArray[-3: 1, 2, 3, 4, 5, 6 ];
in
  array_size( ExampleArray ),
  array_liml( ExampleArray ),
  array_limh( ExampleArray )
end let
```

Note that using `t` defined above,

```
array_size(t)
```

yields 2 and

```
array_size(t[1])
```

yields 32.

6.4 Array Modification

SISAL supports eight intrinsic functions for redefining extant arrays—functions that semantically preserve their arguments. They are described below:

1. `array_addh(A,v)`: generates an array whose value is `V` added to the end of array `A`. The result type is that of `A`.
2. `array_addl(A,v)`: generates an array whose value is `V` added to the front of array `A`, subtracting one from the lower bound. The result type is that of `A`.
3. `array_adjust(A,lo,hi)`: selects a subarray of array `A` between index positions `lo` and `hi` inclusive. The result type is that of `A`, and the resulting lower bound is set to `lo`.
4. `array_setl(A,lo)`: sets the lower bound of array `A` to `lo`. The result type is that of `A`.
5. `array_remh(A)`: removes the last element of array `A`. The result type is that of `A`.
6. `array_reml(A)`: removes the first element of array `A`. The result type is that of `A`, and the new lower bound is one greater than that of `A`.
7. `A||B`: concatenates array `A` and `B`. The result type and lower bound is that of `A`. Both argument arrays must conform.
8. `A[I:V]`: replaces the `I`th component of array `A` with `V`. The result type is that of `A`.

An example use of each operation follows:

```
let
  v1 := array[2: 1.0, 2.0 ];
  v2 := array[3: 3.0, 4.0, 5.0 ];
in

  array_addh(v1,5.0),   % yields array[2: 1.0, 2.0, 5.0 ]
  array_addl(v1,0.0),   % yields array[1: 0.0, 1.0, 2.0 ]

  array_adjust(v2,4,5), % yields array[4: 4.0, 5.0 ]
  array_setl(v2,1),     % yields array[1: 3.0, 4.0, 5.0 ]

  array_reml(v1),       % yields array[3: 2.0 ]
  array_remh(v2),       % yields array[3: 3.0, 4.0 ]

  v1 || v2,             % yields array [2: 1.0, 2.0, 3.0, 4.0, 5.0 ]
                        % the lower bound is the lower bound of v1

  v2[4: 0.0 ],          % yields array[3: 3.0, 0.0, 5.0 ]
  v1[2: 6.0, 7.0 ],     % yields array[2: 6.0, 7.0 ]

  % NOTE THAT THE ABOVE INTRINSICS SEMANTICALLY PRESERVE v1 AND v2, SO
  v1,                   % yields array[2: 1.0, 2.0 ]
```

```
  v2                    % yields array[3: 3.0, 4.0, 5.0 ]

end let
```

6.5 Two Examples

We took the following function from the SIMPLE hydrodynamic benchmark [5]; it illustrates several of the array intrinsics discussed above:

```
% type Twod = array[ array[double_real] ];

function newbz1( klspac:Spac; m:Twod; returns Twod )

%
%       set up boundary zone attributes for matrix m
%
%          lmn-1 lmn lmn+1 lmn=2 lmx-2 lmx-1 lmx lmx+1
%  kmn-1    v     v    v     v     v     v     v    v    ONE
%  kmn      v     E    D     D     D     D     D    H    TWO
%  kmn+1    v     AA   I     I     I     I     I    DD   THREE
%           v     A                                 C    FOUR
%           v     A                                 C
%  kmx-1    v     A                                 C
%  kmx      v     A                                 C
%  kmx+1    v     F    BB    B      B     B     CC  G    SIX
%          lmn-1 lmn lmn+1       lmx-2 lmx-1 lmx lmx+1
%
%

  let
    kmn := klspac.kmn;
    kmx := klspac.kmx;
    lmn := klspac.lmn;
    lmx := klspac.lmx;

    I  := array_adjust(m[kmn+1],lmn+1,lmx);

    AA := m[kmn+1,lmn+1];
    DD := m[kmn+1,lmx];
    BB := m[kmx,lmn+1];
    CC := m[kmx,lmx];

    B  := array_adjust(m[kmx],lmn+2,lmx-1);
    D  := array_adjust(m[kmn],lmn+1,lmx);

    F  := BB;
    H  := DD;
```

```
    G  := CC;
    E  := AA;

    ONE   := array_adjust(m[kmn-1],lmn-1,kmx+1);
    TWO   := array[lmn-1:m[kmn,lmn-1],E]    || D || array[1:H];
    THREE := array[lmn-1:m[kmn+1,lmn-1],AA] || I || array[1:DD];

    FOUR  := for k in kmn+2,kmx
               A := m[k,lmn+1];
               C := m[k,lmx];

               core := array_adjust(m[k],lmn+1,lmx);
               FOUR := array[lmn-1:m[k,lmn-1],A] || core || array[1:C]
             returns array of FOUR
             end for;

    FIVE  := array[lmn-1:m[kmx+1,lmn-1],F,BB] || B || array[1:CC,G];
  in
    array[kmn-1:ONE,TWO,THREE] || FOUR || array[1:FIVE]
  end let
end function
```

The next example defines a collection of routines for initializing and managing a stack of integers:

```
define stack_init, stack_value, stack_push, stack_pop, stack_empty

type stack = array[ integer ];

function stack_init( returns stack )
  array stack []
end function

function stack_value( stack:stack returns integer  )
  stack[ array_limh( stack ) ]
end function

function stack_push( stack:stack; val:integer returns stack  )
  array_addh( stack, val )
end function

function stack_pop( stack:stack returns stack  )
  array_remh( stack )
end function

function stack_empty( stack:stack returns boolean  )
  if ( array_size( stack ) = 0 ) then
    true
```

```
    else
      false
    end if
  end function
```

7 Streams

Like an array, a stream is a one-dimensional collection of homogeneous values. However, differing from an array, one cannot use random integer expressions to index a stream; it must be consumed from beginning to end (like a UNIX pipe). Several simple stream type definitions follow:

```
type OneI = stream[ integer ];      % stream of integer
type OneR = stream[ real ];         % stream of real
type OneD = stream[ double_real ];  % stream of double_real
type OneB = stream[ boolean ];      % stream of boolean
type OneC = stream[ character ];    % stream of character
```

Like arrays, multidimensional streams are possible by declaring stream constituents; that is by defining streams of streams:

```
type TwoI = stream[ OneI ];         % stream of streams of integers
```

Stream type conformity is structural. The following definition conforms to the previous one:

```
type T_TwoI = stream[ stream[ integer ] ];      % similar to TwoI
```

Note that a stream's size is not part of its type declaration. SISAL streams, like SISAL arrays, are dynamic. Their size is defined at run time.

7.1 Simple Stream Definitions

SISAL supports three canonical intrinsics for building streams. The first two are via the `stream of` and `value of catenate` operations discussed in Section 10. The third is via the stream generator. For example,

```
s := stream [ 0.144005d0, 0.177286d0, 0.212769d0, 0.249946d0,
              0.327420d0, 0.366798d0, 0.406054d0, 0.444830d0,
              0.519747d0, 0.555410d0, 0.589629d0, 0.622277d0,
              0.682506d0, 0.709999d0, 0.735729d0, 0.759714d0 ];
```

defines a stream having 16 double real components and a lower bound of one (the arbitrary assignment of a lower bound is not supported). A stream generator, like an array generator, can define an empty stream:

```
s1 := stream OneI [];             % see the definition of OneI above
s2 := stream stream[integer] [];  % s1 and s2 are equivalent
```

The type signature is mandatory.

The following expression defines a stream of streams:

```
cpxS := stream [ stream [ 1, 2, 3, 4 ],
                 stream [ 5, 6, 7, 8 ] ];
```

7.2 Simple Component Selection

SISAL supports two intrinsic operation for selecting stream components. The first allows the components of an entire stream to be scattered across parallel loop iterations (see Section 10). The second intrinsic is `stream_first`. This operation returns the first component of its stream argument. For example,

```
let
  s := stream[ true, false, false ];
in
  stream_first(s)
end let
```

yields `true`. The next example also yields `true`:

```
let
  s := stream[ stream[ true,  false, false ]
               stream[ false, false, false ] ];
in
  stream_first( stream_first(s) )
end let
```

7.3 Stream Inquire

SISAL supports two intrinsic for obtaining the characteristics of a stream: `stream_empty` and `stream_size`. Both functions take a single stream argument of any type. The first returns `true` if the stream is empty. For example,

```
let
  s := stream[ 'a', 'b' ];
in
  stream_empty( s )
end let
```

yields `false`.

The second function, `stream_size`, returns the size of a stream. For example,

```
let
  s := stream[ 0, 1, 2, 3, 4 ];
```

```
  in
    stream_size( s )
  end let
```

yields 5.

7.4 Stream Modification

SISAL supports three intrinsic functions for redefining extant streams—functions that semantically preserve their arguments (like the array modifiers). They are described below:

1. `stream_append(S,v)`: generates a stream whose value is `V` added to the end of stream `S`. The result type is that of `S`.
2. `array_rest(S)`: generates a stream whose value is stream `S` without its first element. The new lower bound is one.
3. `S1||S2`: concatenates stream `S1` and `S2`. The result type is that of `S1`. Both argument streams must conform. The new lower bound is one.

An example use of each operation follows:

```
  let
    s1 := stream[ 1.0, 2.0 ];
    s2 := stream[ 3.0, 4.0, 5.0 ];
  in

    stream_append(s1,6.0), % yields stream[ 1.0, 2.0, 6.0 ]
    stream_rest(s2),       % yields stream[ 4.0, 5.0 ]

    s1 || s2,              % yields stream[ 1.0, 2.0, 3.0, 4.0, 5.0 ]

    % NOTE THAT THE ABOVE INTRINSICS SEMANTICALLY PRESERVE s1 AND s2:
    s1,                   % yields stream[ 1.0, 2.0 ]
    s2                    % yields stream[ 3.0, 4.0, 5.0 ]

  end let
```

8 Records

SISAL records are similar to PASCAL records and C structures. They define collections of possibly heterogeneous values. Several example record type definitions follow:

```
type materialrec = record[ mtl:integer;
                           atrat,
                           dns,
```

```
                              tmp:double_real;
                              matb,mate:integer ];

type input = record[ igroup,irr,ithom,
                     ixopec,naxl,nradl:integer;
                     axl,bwgt,
                     opec,radl,
                     tcen,wcut,xmult:double_real;
                     mtls: array[materialrec] ];

type coordrec = record[ rr,zz:double_real ];
```

Two record types conform if the number, order, and type of their respective components conform. For example,

```
   R1 := record[ a,b,c:real; d:array[integer] ]
```

and

```
  CompType := real;
  OneDimI  := array[integer];

  R2 := record[ e,f,g:CompType; e:OneDimI ]
```

conform, but `R3` shown below does not:

```
  R3 := record[ d:array[integer]; a,b,c:real ];
```

Field names need not match.

8.1 Record Definitions

SISAL supports a single generator for defining a record. An example follows:

```
  R := record [ e:2.71; pi:3.14 ];
```

Semicolons separate the field definitions. Note that `R` is not equivalent to `Rprime` defined below. The two values do not conform in type.

```
  Rprime := record [ pi:3.14; e:2.71 ];
```

8.2 Component Selection

Record component selection is conventional, using dot notation. Given `R` defined above, `R.pi` yields `3.14`.

8.3 Record Modification

SISAL supports a single operation for modifying a record. It is similar in functionality to the array replace operation. For example,

```
let
  Smart := record[ data:record[ name:"Unknown"; age:0 ]; gpa:4.0 ];
in
  Smart replace[ data:record[ name:"Sue"; age:28 ] ]
end let
```

yields

```
record [ data:record[ name:"Sue"; age:28 ]; gpa:4.0  ]
```

Note that `replace` is a reserved word.

8.4 A Common Use for Records

The following example uses the record data type and its intrinsic operations to define a complex number package. The routines were taken from a weather modeling code developed at RMIT in Australia:

```
define Cadd, Csub, Cmul, Cdiv

type CplexReal = Record[Repart,Impart:real];

% cnum1 + cnum2
function Cadd( cnum1,cnum2:CplexReal returns CplexReal )
  record [ Repart: cnum1.Repart + cnum2.Repart;
           Impart: cnum1.Impart + cnum2.Impart ]
end function

% cnum1 - cnum2
function Csub( cnum1,cnum2:CplexReal returns CplexReal )
  record [ Repart: cnum1.Repart - cnum2.Repart;
           Impart: cnum1.Impart - cnum2.Impart ]
end function

% cnum1 * cnum2
function Cmul( cnum1,cnum2:CplexReal returns CplexReal )
  record [ Repart: cnum1.Repart * cnum2.Repart -
                   cnum1.Impart * cnum2.Impart;
           Impart: cnum1.Repart * cnum2.Impart  +
                   cnum1.Impart * cnum2.Repart  ]
end function
```

```
% cnum1/cnum2
function Cdiv( cnum1,cnum2:CplexReal returns CplexReal )
  let
    dnom := cnum2.Repart * cnum2.Repart +
            cnum2.Impart * cnum2.Impart;
  in
    record [ Repart: (cnum1.Repart*cnum2.Repart +
                      cnum1.Impart*cnum2.Impart) / dnom;
             Impart: (cnum1.Impart*cnum2.Repart -
                      cnum1.Repart*cnum2.Impart) / dnom ]
  end let
end function
```

9 Unions

A SISAL union defines a single value from a collection of values, each possible of different type. SISAL strictly manages the access to unions to prevent value aliasing and the other trickery that prevents program correctness and portability in imperative programs. Some example union type declarations follow:

```
type Stuff = union[ A:integer; B:real;
                    C:stream[real];
                    D,E,F:double_real ];

% an enumerated type!
type Quadrant = union[ NE,SE,SW,NW:null ];

% a recursive type definition!
type QTree = union[ Scalar:real;
                    NonScalar:record[ NE,SE,SW,NW:QTree ] ];
```

The `null` type is special in that SISAL does not support operations to manipulate values in its domain. In fact, the domain contains only one value: `nil`. The null type is most often used to define enumerated types or ground recursive union definitions. Note that only union types allow recursive definitions. We refer to the union fields as tags. Union type conformity rules are those of records (see Section 8).

9.1 Union Definitions

A single generator is supported to define union values. Differing from records, only one field may be defined. Further, a type signature is required:

```
% this definition uses the type name defined in the previous example
U := union Stuff [ C:stream[ 1.0, 2.0, 3.0, 4.0 ] ];
```

9.2 Component Selection

Union component selection syntax is not familiar; that is, SISAL does not use dot notation. Instead one must use a `tagcase` construct. Like the `if` construct, the arms of the tagcase must agree in arity and conform in type. The following expression illustrates how one would access the component of union `U`, which is assumed to be of type `Stuff`:

```
% NOTE THAT EACH ARM DEFINES A double_real VALUE
tagcase V := U
  tag A:
    4.0D0
  tag D,E:
    V+4.0D0
  tag C:
    double_real( stream_first( V ) )
  otherwise:
    0.0D0
end tagcase
```

Here, if the active tag of `U` is `C`, then the third alternative defines the result. Observe that the defined value itself is available to the selected arm via the name bound in the head of the construct. The otherwise pattern is optional; however, it is required if all the tags are not enumerated. For example, the following expression is equivalent to the previous one:

```
tagcase V := U
  tag A:
    4.0D0
  tag D,E:
    V+4.0D0
  tag C:
    double_real( stream_first( V ) )
  tag B,F:
    0.0D0 % NOTE V CANNOT BE REFERENCED IN THIS CONTEXT AS THE TAG
          % TYPES CONFLICT: real and double_real
end tagcase
```

9.3 Union Inquiries

SISAL supports an intrinsic (with nonstandard syntax) for identifying the active tag of a union. The operation yields boolean `true` if the specified tag is active; for example,

```
let
  U := union Stuff[ A:-3 ];
in
  is C(U)
end let
```

yields `false` as `A` is the active tag.

9.4 A Simple Example

The following example program converts an array of integers into a binary tree using the union data type and its operations.

```
define Main

type OneDimI = array[ integer ];

type BTree = union[ Data: integer;
                    NonData: record[ L,R:BTree ] ];

% Main defines a BTree representation of the input array A

function Main( A: OneDimI returns BTree )
  if array_size( A ) = 1 then
    union BTree[ Data: A[ array_liml( A ) ] ]
  else
    let
      Mid := array_liml( A ) + array_size( A ) / 2;
      L := Main( array_adjust( A, array_liml( A ), Mid-1 ) );
      R := Main( array_adjust( A, Mid, array_limh( A ) ) )
    in
      tagcase L1 := L
        tag NonData:
          union BTree[ NonData: record[ L:L; R:R ] ]
        tag Data:
          tagcase R1 := R
            tag NonData:
              union BTree[ NonData: record[  L:L; R:R ] ]
            tag Data:
              if L1=R1 then
                L
              else
                union BTree[ NonData: record[ L:L; R:R ] ]
              end if
          end tagcase
      end tagcase
    end let
  end if
end function
```

10 Product Form Loops

SISAL supports two constructs for describing iterative definitions: `for` and `for initial`. The first is for parallel definitions and the second is for sequential definitions. In this section, we illustrate and discuss the parallel or product form, which cannot introduce carried values (enforced by semantics). As with SISAL's other compound constructs, the `for` itself is an expression that can appear within another expression.

The `for` has three parts: top, body, and returns. The top defines a sequence of index values, one for each body execution. The values defined in each executed body are then passed to the returns for packaging. The order of packaging, unless specified otherwise, is semantically equivalent to a left-to-right consideration of the original sequence of index values. A simple example follows:

```
for i in 1,n              % note: this example does not have a body
returns value of sum 1.0
end for
```

Here for each value bound to `i` (integers between `1` and `n` inclusive) `1.0` is added to a global counter; hence, yielding `real(n)`. Now consider

```
for j in 1,n
  pval := A[pivot,pivot];  % select the pivot value
  v := row[j]/pval;        % this semicolon is optional
returns array of v
end for
```

which processes the current pivot row of a Gaussian Reduction. In this example the body is not empty, and has a syntax and semantics identical to the `let` clause of a `let` expression. The above expression, assuming `n` represents `3`, is equivalent to

```
let
  pval := A[pivot,pivot];  % select the pivot value

  v1 := row[1]/pval;
  v2 := row[2]/pval;
  v3 := row[3]/pval;
in
  array[1: v1, v2, v3 ]
end let
```

10.1 Result Packaging

SISAL supports a variety of operations for packaging results. The various packaging operations are:

1. `array of`: for packaging an array.
2. `stream of`: for packaging a stream.
3. `value of`: to return the last value.

4. `value of` reduction_op: for reducing a sequence. A variety of reduction operations are supported:

   (a) `sum`: for summation

   (b) `product`: for multiplication

   (c) `greatest`: for finding maximums

   (d) `least`: for finding minimums

   (e) `catenate`: for concatenating arrays and streams.

The compound example below demonstrates each of the above operations and documents their functionality:

```
for i in 1,3
returns array of i                  % yields array[1: 1, 2, 3 ]
        stream of i                 % yields stream[ 1, 2, 3 ]
        value of i                  % yields 3
        value of sum i              % yields 1+2+3 or 6
        value of product i          % yields 1*2*3 or 6
        value of least i            % yields min(min(1,2),3) or 1
        value of greatest i         % yields max(max(1,2),3) or 3
        value of catenate array[1:i]  % yields array[1: 1, 2, 3 ]
end for
```

Note that the clauses are not comma separated.

10.2 Filters

Any one of the above operations can provide a filter to eliminate values. For example, the following expression yields an array of odd integers:

```
for i in 1,n
returns array of i when mod(i,2) ~= 0
end for
```

An equivalent coding is:

```
for i in 1,n
returns array of i unless mod(i,2) = 0
end for
```

10.3 Dot Products

A top may introduce two or more sequences via the `dot` operation. For example,

```
for i in 1,4 dot j in 5,8
returns array of i+j
end for
```

yields

```
array[1: 6, 8, 10, 12 ]   % from array[1: 1+5, 2+6, 3+7, 4+8 ]
```

10.4 Cross Products

A top may introduce a cartesian or outer product of sequences, which defines an implicit nesting of `for` expressions. For example,

```
for i in 1,2 cross j in 1,2
returns array of 1.0/real(i+j-1)
end for
```

defines a hilbert matrix which is equivalent to

```
for i in 1,2
  Row := for j in 1,2
         returns array of 1.0/real(i+j-1)
         end for;
returns array of Row
end for
```

and

```
array[1: array[1: 1.0/1.0, 1.0/2.0 ],
         array[1: 1.0/2.0, 1.0/3.0 ] ];
```

10.5 Array and Stream Scattering

For convenience, the top of a `for` expression can directly scatter an array or stream. For example,

```
for x in array[1: 100.0, 200.0, 300.0, 400.0 ]
returns value of sum x
end for
```

is the same as

```
let
  A := array[1: 100.0, 200.0, 300.0, 400.0 ]
in
  for i in array_liml(A), array_limh(A)
```

```
    returns value of sum A[i];
    end for
  end let
```

which is the same as 1000.0.

The top can define the corresponding sequence of index positions for the scattered values:

```
  % I is 1,2,3, and 4 respectively (see the definition of A above)
  for x in A at I
  returns array of x*B[I]
  end for
```

10.6 Common Mistakes

The most common mistake with `for` expressions concerns the defined bounds of packaged results. The following expression should clear the confusion; it yields 2 and 1 respectively:

```
  function bounds_test( returns integer, integer )
    let
      % NOTE: a WILL HAVE A LOWER BOUND OF 2
      a := for i in 2,3 returns array of i end for;

      % NOTE: b WILL HAVE A LOWER BOUND OF 1, NOT 4!!!
      b := for i in 4,6 returns value of catenate a end for;
    in
      array_liml(a), array_liml(b)
    end let
  end function
```

11 Sequential Loops

SISAL's sequential or non-product loop defines sequential iteration. An example follows; it approximates the square-root of `X`:

```
  for initial
    Root := X/2.0D0
  while abs( Root*Root - X ) > Epsilon repeat
    Root := ( X/old Root + old Root ) / 2.0D0
  returns value of Root
  end for
```

Note the use of `old` to reference values defined on the previous iteration; that is,

```
  Root := ( X/Root + Root ) / 2.0D0
```

is illegal SISAL.

As seen, the `for initial` loop has four parts: init, test, body, and returns. The init defines the first values passed to the returns and defines the initial value of each carried item. The test verifies the conditions for continued iteration, and each body invocation defines the next value passed to the returns, and establishes the next set of carried values. The test can comprise any legal boolean expression, and the returns is similar in syntax and semantics to that of the `for` (see Section 10).

To get a better feel for its overall semantics, consider the following loop

```
for initial
  i := 1;                % establishes the carried value
  v := i;                % this defines the result's first component
while ( i <= 2 ) repeat
  i := old i + 1;
  v := i + old i;        % the same as (old i + 1) + old i
returns array of v
end for
```

which defines

```
array[1: 1, 3, 5 ];
```

Note that the binding of `i` to `old i` takes place just before the body completes. Also note that the implicit first iteration defines the first component of the result (`1` in this case). The following loop is equivalent to the one shown above, but has an inverted test:

```
for initial
  i := 1;
  v := i;
until ( i > 2 ) repeat    % note the until reserved word
  i := old i + 1;
  v := i + old i;
returns array of v
end for
```

11.1 Post-Test Loops

SISAL supports post-test loops as well as the pre-test loops. Consider the following recoding of the previous example. The returns is guaranteed to execute at least twice:

```
for initial
  i := 1;
  v := i;
repeat
  i := old i + 1;
```

```
    v := i + old i;
  while ( i <= 2 )
  returns array of v
  end for
```

A version using the `until` test follows:

```
  for initial
    i := 1;
    v := i;
  repeat
    i := old i + 1;
    v := i + old i;
  until ( i > 2 )
  returns array of v
  end for
```

11.2 Common Mistakes

Like the `for`, the most common mistake with `for initial` expressions concerns the bounds of packaged arrays. The following expression should clear the confusion; it yields 1 and 1 respectively:

```
  let
    % NOTE: c WILL HAVE A LOWER BOUND OF 1
    c := for initial
           i := 2;
         while ( i < 3 ) repeat
           i := old i + 1;
         returns array of i
         end for;

    % NOTE: d WILL HAVE A LOWER BOUND OF 1
    d := for initial
           i := 2;
         while ( i < 3 ) repeat
           i := old i + 1;
         returns value of catenate c
         end for;
    in
     array_liml(c), array_liml(d)
    end let
  end function
```

An additional mistake concerns confusion over the implicit first iteration. Regardless of context, the init defines the first iteration. For example,

```
for initial
  i := 1;
  v := 4.0;
while ( i < n ) repeat
  i := old i + 1;
  v := 2.0;
returns value of sum v
end for
```

yields 4.0 if the body never executes; that is, if `n <= 1`.

12 FIBRE

FIBRE is used to format textual input and output from SISAL programs. One can find a complete definition in [7]. Here we simply illustrate the data formats and identify the pertinent features. All of SISAL's data values can be formatted using FIBRE.

12.1 Line Formats and Comments

Like SISAL, FIBRE is free format. Multiple occurrences of "white space" (blanks, tabs, and newlines) are allowed between values and delimiters. Comments are a special form of white space and have syntax similar to SISAL but not identical:

```
# THIS IS A FIBRE COMMENT AS IT BEGINS WITH A POUND CHARACTER
```

Note that comments, strings and simple values cannot cross line boundaries.

12.2 Simple Value Formats

Simple FIBRE values have the same syntax as SISAL literals. An example of each follows:

```
# integer values
0
2010
-32

# real and double real values
34d0        # double real
34.0E0      # real
0.14159     # real
301.4e-64   # real
-302.4d64   # double real

# character values
'\n'        # newline
```

```
'a'         # a
'\\'        # backslash
'\001'      # control-a

# boolean values
T           # true
F           # false

# null values
nil
```

12.3 Aggregate Value Formats

In general, the aggregate value syntax of FIBRE differs from SISAL. We consider each of the possible formats below.

12.3.1 Arrays

FIBRE array values describe data of type array. Unless they take the form of a string, a lower bound must be provided (an optional upper bound is supported by most FIBRE implementations, including the Optimizing SISAL Compiler [3]). Examples follow:

```
# array[integer] without upper bound specifications
[1: 1 2 3 4 5 6 ]
[1:]                # an empty array

# array[integer] with upper bound specifications
[1,6: 1 2 3 4 5 6 ]
[1,0:]              # an empty array

# array[ array[reals] ]  without upper bound specifications
[1:
  [1: 1.0 2.0 3.0 0.0 ]
  [1: 4.0 1.0 6.0 0.0 ]
  [1: 4.0 5.0 1.0 0.0 ]
]

# array[ array[reals] ]  with upper bound specifications
[1,3:
  [1,4: 1.0 2.0 3.0 0.0 ]
  [1,4: 4.0 1.0 6.0 0.0 ]
  [1,4: 4.0 5.0 1.0 0.0 ]
]
```

Note the absence of comma separators and the `array` reserved word.

12.3.2 Streams

FIBRE stream values describe data of type stream. Examples follow. Again, comma separators are not used:

```
# stream[boolean]
{
  T T F T F F F
  F F F T T T T
  T T F T F T F
}

# stream[ array[real] ]
{
  [1,2: 1.0 2.0 ]
  [3: 3.0 -0.123E-4]
}
```

12.3.3 Records

FIBRE record values describe data of type record. One does not provide field names or comma separators. Several examples follow:

```
# type complex = record[ z:real; y:real ]
< 1.23 0.0 >     # z followed by y

# array[ complex ]
[1,4:
  < 0.0 0.0 >
  < 1.0 0.0 >
  < 1.0 0.0 >
  < 0.0 0.0 >
]
```

12.3.4 Unions

FIBRE union values describe data of type union. One does not provide the symbolic tags, however associated ordinal values must be provided. The ordinals begin at zero, and are assigned in the order in which that tags appear in the associated type. Consider the following example, which defines a list:

```
# type List = union[ Empty: null;
#                    Occupied: record[ data:integer; next:List ];

# the ordinal of Empty is 0 and Occupied is 1.
```

```
# a list of two values: 100 followed by 200
( 1:
  < 100
    ( 1:
      < 200 ( 0:nil ) >
    )
  >
)
```

13 Example Programs

In this section, we present the SISAL source for 17 programs, ranging from Factorial to Gaussian Elimination and the simulation of Gel Chromatography. In several cases, we provide both sequential and parallel codings. For the largest program (Gel Chromatography), we present the FORTRAN from which we developed the SISAL.

13.1 Factorial

```
% ****************************************************************
define Main

%
% Main(4) -> 4*3*2*1
%

function Main( n:integer returns integer )
  if ( n <= 0 ) then 1 else n*Main(n-1) end if
end function
```

13.2 Matrix Multiply

```
% ****************************************************************
define Main

%
% Main(2,3) ->
%
%   A = 3.0 4.0 5.0       B = 4.0 5.0
%       4.0 5.0 6.0           5.0 6.0
%                             6.0 7.0
%
%   AxB = 62.0 74.0
%         77.0 92.0
%
```

```
type OneDim = array[ real ];
type TwoDim = array[ OneDim ];

function Matmult( A,B:TwoDim;  M,N,L:integer returns TwoDim )
  for i in 1, M cross j in 1, L
    S := for k in 1, N
         returns value of sum  A[i,k]*B[k,j]
         end for
  returns array of S
  end for
end function

function Main( M,N:integer returns TwoDim,TwoDim,TwoDim )
  let
    % BUILD TEST MATRIX A (M by N)
    A := for i in 1,M cross k in 1,N
         returns array of 1.0+real(i+k)
         end for;

    % BUILD TEST MATRIX B (N by M)
    B := for k in 1,N cross i in 1,M
         returns array of 2.0+real(k+i)
         end for;

    AxB := Matmult( A,B, M,N,M );
  in
    A,B,AxB
  end let
end function
```

13.3 Matrix Transpose

```
% *****************************************************************
define Main

%
% main(2,3) ->
%
%   A := 2 3 4     Atran :=  2 3
%        3 4 5               3 4
%                            4 5
%

type OneDim = array[ integer ];
type TwoDim = array[ OneDim ];

function Transpose( A:TwoDim returns TwoDim )
```

```
  let
    N := array_size(A);
    M := array_size(A[1]);
  in
    for i in 1,M cross j in 1,N
    returns array of A[j,i]
    end for
  end let
end function

function Main( N,M:integer returns TwoDim,TwoDim )
  let
    % BUILD TEST MATRIX A (N by M)
    A := for j in 1,N cross i in 1,M
         returns array of i+j
         end for;

    Atran := Transpose(A);
  in
    A,Atran
  end let
end function
```

13.4 Calculating Pi: Sequential Version

```
% *****************************************************************
define Main

%
% Main(Cycles) yields a very slowly converging approximation to pi
%              using the alternating series 1 - 1/3 + 1/5 - 1/7 ...
%
%
% Main(10000) -> 3.141797
%

function Main ( Cycles:integer returns real )
  4.0 * for initial
          Approx := 1.0; Sign := 1.0;
          Count  := 1
        while ( Count < Cycles ) repeat
          Sign   := -old Sign;
          Count  := old Count+2;
          Approx := old Approx+Sign/real(Count)
        returns
          value of Approx
        end for
```

```
end function
```

13.5 Calculating Pi: Parallel Version

```
% ************************************************************
define Main

%
% Main(Cycles) yields an approximation to pi using
%         (4/N)*sum(j=1,N)(1/(1+x[j]^2), where x[j] = (j-1/2)/N.
%         See Karp and Babb, "A comparison of 12 parallel FORTRAN
%         dialects," IEEE Software (September 1988):  52-67.
%
% Main(10000) -> 3.141591
%

function Main ( Cycles:integer returns real )
  (4.0/real(Cycles)) * for j in 1,Cycles
                         x := (real(j)-0.5)/real(Cycles)
                       returns value of sum 1.0/(1.0+x*x)
                       end for
end function
```

13.6 Harmonic Mean, Etc.

```
% ************************************************************
define Main

%
% Stats(data) -> defines the harmonic and arithmetic means and
%                minimum and maximum values
%
% data := 2.0, 4.45, 10.9, 3.3, 2.0, 2.0, 1.0, 16.0
%
%   harm := 2.51414
%   avg  := 5.206
%   minv := 1.0
%   maxv := 16.0
%

type double = double_real;
type OneDim = array[ double ];

function Stats( data:OneDim returns double,double,double,double )
  let
    num := double_real(array_size(data));
```

```
    denom, maxv,
    minv, total  := for x in data
                      returns value of sum 1.0D0/x
                              value of greatest x
                              value of least x
                              value of sum x
                      end for;

    harm  := num/denom;
    avg   := total/max(num,1.0D0)
  in
    harm, avg,
    minv, maxv
  end let
end function

function Main( returns double,double,double,double )
  let
    data := array[1:  2.0D0, 4.45D0,
                     10.9D0,  3.3D0,
                      2.0D0,  2.0D0,
                      1.0D0, 16.0D0 ];
  in
    Stats( data )
  end let
end function
```

13.7 Sieve of Eratosthenes

```
% ***************************************************************
define Main

%
% Generates the primes between 2 and Limit+2 inclusive.
%
%  Main(20) -> 2 3 5 7 11 13 17 19
%

type StrmInt = stream[ integer ];

global sqrt( x:double_real returns double_real )

function Filter( S:StrmInt; M:integer returns StrmInt )
  for I IN S returns stream of I unless mod(I,M) = 0 end for
end function
```

```
 function Integers( Limit:integer returns StrmInt )
   for initial
     I := 3;
   while I <= Limit repeat
     I := old I + 2;
   returns stream of I
   end for
end function

function main( Limit:integer returns StrmInt )
  let
    Maxt := integer( sqrt(double_real(Limit)) );
  in
    for initial
      S := Integers(Limit);
      T := 2;
    repeat
      T := stream_first(old S);
      S := if T <= Maxt then
             Filter( stream_rest(old S), T )
           else
             stream_rest(old S)
           end if
   until stream_empty( S )
   returns stream of T
   end for
 end let
end function
```

13.8 Word Count

```
% ****************************************************************
define Main

%
% Main(text) -> counts the number of words in string text
%
% Main( "This sentence has Five words" ) -> 5
%

type String = Array [ character ];

function is_char( c:character returns boolean )
  if ( (c = ' ') | (c = '\t') | (c = '\n') ) then
    false
  else
    true
```

```
  end if
end function

function Main( text:String returns integer )
  for initial
    count := 0; iword := 0; pointer := 1;
  while (pointer <= array_size(text)) repeat
    pointer := old pointer + 1;

    count,
    iword :=
      if ( is_char(text[old pointer]) & (old iword = 0)) then
        1,1
      elseif ( is_char(text[old pointer]) & (old iword = 1)) then
        0,1
      else
        0,0
      end if;
   returns value of sum count
   end for
end function
```

13.9 Index of First Minimum: Sequential Version

```
% ******************************************************************
define  Main

%
% ifmin(X) Finds the location of first minimum in array X
%
%   X := -1.0 -22.9 33.349 33.2 66.1 -22.9 0.332
%
%   ifmin :=  2
%

type double = double_real;
type OneDim = array[ double ];

function ifmin( X:OneDim returns integer )
  for initial
    imin := 1; k := 2;
  while ( k <= array_size(X) ) repeat
    k := old k + 1;

    imin := if ( X[old k] < X[old imin] ) then
              old k
            else
```

```
                old imin
              end if;
    returns value of imin
    end for
  end function

  function Main( returns OneDim, integer )
    let
      X := array[1: -1.0D0, -22.9D0, 33.349D0, 33.2D0,
                    66.1D0, -22.9D0, 0.332D0 ]
    in
      X, ifmin( X )
    end let
  end function
```

13.10 Index of First Minimum: Parallel Version

```
% ******************************************************************
define  Main

%
% ifmin(X) Finds the location of first minimum in array X
%
%   X := -1.0 -22.9 33.349 33.2 66.1 -22.9 0.332
%
%   ifmin :=  2
%

type double = double_real;
type OneDim = array[ double ];

function ifmin( X:OneDim returns integer )
  let
    vmin := for Elm in X returns value of least Elm end for;
  in
    for Elm in X at I
    returns value of least I when Elm = vmin
    end for
  end let
end function

function Main( returns OneDim, integer )
  let
    X := array[1: -1.0D0, -22.9D0, 33.349D0, 33.2D0,
                  66.1D0, -22.9D0, 0.332D0 ]
  in
    X, ifmin( X )
```

```
  end let
end function
```

13.11 Batcher Sort

```
% *************************************************************
define Main

%
%  List  := 99 100 44 33 2 -4 109 3 -222 15 888 -4 -4 3 22
%  SList := -222 -4 -4 -4 2 3 3 15 22 33 44 99 100 109 888
%

type OneDim = array[ integer ]

function CeilingOfLog2( N:integer returns integer )
  for initial
    L := 0;
    TwoToTheL := 1
  while TwoToTheL < N repeat
    L := old L + 1;
    TwoToTheL := 2*old TwoToTheL
  returns value of L
  end for
end function

function FloorOfNOver2( N: integer returns integer )
  if N=1 then 0 else N/2 end if
end function

function Isec( I,P:integer returns integer )
  if mod( I/P, 2 ) = 1 then P else 0 end if
end function

function Batcher( K:OneDim returns OneDim )
  let
    N := array_size(K);
    T := CeilingOfLog2( N )
  in
    for initial
      P := exp( 2, T );
      B := array_setl( K, 0 )
    while P>1 repeat
      P := FloorOfNOver2( old P );
      B :=
        for initial
          Q := exp( 2, T );
```

```
          R := 0;
          D := P;
          C := old B
        repeat
          C :=
            for Elt in old C at I
              NewElt :=
                if (Isec(I,P) = old R) & (I+old D < N) then
                  min( Elt, old C[ I+old D ] )
                elseif ( Isec(I-old D,P) = old R) & (I >= old D) then
                  max( Elt, old C[ I-old D ] )
                else
                  Elt
                end if;
            returns array of NewElt
            end for;
          D := old Q - P;
          Q := old Q / 2;
          R := P
        until Q<P
        returns value of C
        end for
    returns value of B
    end for
  end let
end function

function Main( returns OneDim, OneDim )
  let
    List  := array[1: 99, 100, 44, 33, 2, -4,
                      109, 3, -222, 15, 888,
                      -4, -4, 3, 22 ];
    SList := Batcher(List);
  in
    List, SList
  end let
end function
```

13.12 Gaussian Elimination

```
% *****************************************************************
define Main

%
% Gauss-Jordan elimination with no pivoting, based on the  routine
% given in the book "Numerical Recipes-The Art of Scientific
% Computing" (Press, Flannery, Teukolsky, Vetterling) pages 28-29.
```

```
% Ain is an input matrix of NxN double_real floating point numbers and
% Bin is a vector of N double_real floating point numbers. The program
% returns the vector x in the equation A x = B.
%
% Main( n,Ain,Bin ), where
%
%   n := 4
%
%   Ain := 1.0 1.0 1.0 1.0
%          1.0 2.0 3.0 4.0
%          1.0 4.0 9.0 16.0
%          1.0 8.0 27.0 64.0
%
%   Bin := 1.0 0.0 0.0 0.0
%
%   gives 4.0 -6.0 4.0 -1.0
%

type double = double_real;
type OneI   = array[integer];
type OneD   = array[double];
type TwoD   = array[OneD];

function Reduce( n,pivot:integer; A:TwoD; B:OneD returns TwoD, OneD )
  for i in 1,n
    row  := A[i];
    mult := A[i,pivot]/A[pivot, pivot];
    rA,
    rB := if i = pivot then
            for j in 1,n
            returns array of row[j]/A[pivot,pivot]
            end for,
            B[i]/A[pivot, pivot]
          else
            for j in 1,n
            returns array of row[j]-mult*A[pivot,j]
            end for,
            B[i]-mult*B[pivot]
          end if
  returns array of rA
          array of rB
  end for
end function

function Main( n:integer; Ain:TwoD; Bin:OneD returns OneD )
  for initial
    i    := 0;
    A, B := Ain, Bin;
```

```
  while i < n repeat
    i    := old i + 1;
    A, B := Reduce( n,i, old A, old B );
  returns value of B
  end for
end function
```

13.13 Random Number Package

```
% ***********************************************************
define rans, ranf

%
% This is the SISAL version of the RANF random number generator.
% Rans accepts a sequence number and initial seed, and divides the
% RANF sequence into that number of subsequences, each of which can
% be sampled in parallel, by Ranf (which takes the previous seed in
% the sequence as input).
%

type Four_Plex = array[ integer ];  % The four 12-bit integers that
                                    % when used together simulate a
                                    % 48-bit integer or the mantissa
                                    % of a 64-bit floating point
                                    % number.

type Seed_Array = array[ Four_Plex ]; % One seed per process.
type Bit_Array  = array[ integer ];   % The binary expansion of
                                      % a Four_Plex.

type double = double_real;


%==============================================================
% F O R W A R D   F U N C T I O N   D E C L A R A T I O N S
%==============================================================

forward function ranf( Seed:Four_Plex returns double, Four_Plex )
forward function rans( N,Seed1:integer returns Seed_Array )
forward function ranf_a_to_k( a:Four_Plex; k:Bit_Array
                              returns Four_Plex)
forward function ranf_even( n:integer returns integer )
forward function ranf_k( n:integer returns Four_Plex )
forward function ranf_k_binary( k:Four_Plex returns Bit_Array )
forward function ranf_mod_mult( a,b:Four_Plex returns Four_Plex )
forward function ranf_odd( n:integer returns integer )
```

```
%==============================================================
% F U N C T I O N   D E F I N I T I O N S
%==============================================================

function ranf( Seed:Four_Plex returns double, Four_Plex )

% Returns the next random number and seed in the series. The seed's
% bits shifted to the right 48 places are the bits  of  the  random
% number.  The next seed is the product of the current seed and the
% series' multiplier.

  double_real(Seed[3]) / 4096.0d0 +
  double_real(Seed[2]) / 16777216.0d0 +
  double_real(Seed[1]) / 68719476736.0d0 +
  double_real(Seed[0]) / 281474976710656.0d0,
  ranf_mod_mult( array [0: 373, 3707, 1442, 647], Seed )
end function


function rans( N_In, Seed1 : integer returns array[ Four_Plex ] )

% Returns n seeds by dividing the wheel  into  n  segments  and
% returning the start seed of each segment.  If seed1 is 0, the
% routine will return all zeros; in such cases, seed1 is set to
% [3281, 4041, 595, 2376] which is statistically the best start-
% ing seed.  The wheel is divided into n pieces by dividing its
% period (2**46) by n ( = k ). Then
%
%     seed(i) = seed(i - 1) * a ** k  (mod 2**48)
%
% where a is the multiplier of the random number generator.

% E M B E D D E D   F U N C T I O N
%----------------------------------------------
function N_Is_Odd( N : integer returns Boolean )
if mod( N, 2 ) = 1 then true else false end if
end function % N_Is_Odd
%----------------------------------------------

  for initial

    N       := if N_is_Odd( N_In) then N_In else N_In + 1 end if;
    i       := 1;
    seed    := if Seed1 = 0 then
                 array [0: 3281, 4041, 595, 2376]
               else
                 array [0: abs(Seed1), 0, 0, 0]
               end if;
```

```
    a       := array [0: 373, 3707, 1442, 647];
    a_k     := if N > 1 then
                 ranf_a_to_k( a, ranf_k_binary( ranf_k( N ) ) )
               else % Don't bother to raise a to the first power.
                 a
               end if

  while i < N repeat
    i    := old i + 1;
    seed := ranf_mod_mult(old seed, a_k)
  returns array of seed
  end for
end function

function ranf_a_to_k( a:Four_Plex; k:Bit_Array returns Four_Plex )

% Returns a to the k mod 2**48.   k is a binary number, so it is
% equivalent to the sum of k[i] * 2**i.  a to the k mod 2**48 is
% thus the product of a * (k[i] * 2**i) mod 2**48.  We calculate
% iteratively a ** (2**i) mod 2**48, and form the result by mul-
% tiplying together those terms (mod 2**48) for which k[i] is 1.

  for initial
    i   := 0;
    a_i := a;
    a_k := array [0: 1, 0, 0, 0]
  while i < 46 repeat
    i   := old i + 1;
    a_k := if k[i] = 0 then
             old a_k
           else
             ranf_mod_mult( old a_k, old a_i )
           end if;
    a_i := ranf_mod_mult( old a_i, old a_i )
  returns value of a_k
  end for
end function


function ranf_even( n:integer returns integer )
  if mod( n, 2 ) = 0 then 1 else 0 end if
end function

function ranf_k( n: integer returns Four_Plex )

  % Returns the floor of (2**46) / nn, where 2**46 is represented by
  % 4 12-bit integers, in base ten [1024, 0, 0, 0], and the division
  % is done piecemeal. For statistical reasons the divisor is made
```

```
  % odd.
  %
  let
    nn := n + ranf_even( n );

    q3 := 1024 / nn;
    r3 := 1024 - (nn * q3);
    q2 := (r3 * 4096) / nn;
    r2 := (r3 * 4096) - (nn * q2);
    q1 := (r2 * 4096) / nn;
    r1 := (r2 * 4096) - (nn * q1);
    q0 := (r1 * 4096) / nn
  in
    array [0: q0, q1, q2, q3]
  end let
end function


function ranf_k_binary( k:Four_Plex returns Bit_Array )

  % Returns the binary representation of k, where k is represented
  % by 4 12-bit integers. The bits are stored in the returns array
  % from least significant to most significant.

  for i in 0, 3
  returns value of catenate

      for initial
        j   := 1;
        x   := k[i] / 2;
        bit := ranf_odd( k[i] )
      while j < 12 repeat
        j   := old j + 1;
        x   := old x / 2;
        bit := ranf_odd( old x )
      returns array of bit
      end for
  end for
end function


function ranf_mod_mult( a,b:Four_Plex returns Four_Plex )

  % Returns a * b where both a and b are 4 12-bit integers and the
  % multiplication is done piecemeal.

  let
    j0 := a[0] * b[0];
```

```
    j1 := a[0] * b[1] + a[1] * b[0];
    j2 := a[0] * b[2] + a[1] * b[1] + a[2] * b[0];
    j3 := a[0] * b[3] + a[1] * b[2] + a[2] * b[1] + a[3] * b[0];
    k0 := j0;
    k1 := j1 + k0 / 4096;
    k2 := j2 + k1 / 4096;
    k3 := j3 + k2 / 4096
  in
    array [0: mod(k0, 4096), mod(k1, 4096),
              mod(k2, 4096), mod(k3, 4096)]
  end let
end function


function ranf_odd( n:integer returns integer )
  if mod( n, 2 ) = 0 then 0 else 1 end if
end function % ranf_odd
```

13.14 The Game of Life

```
% ****************************************************************
define Main

%
% John Conway's Game of Life. Values of the Grid are 0's or 1's.  A
% given cell has 8 neighbors.  One each iteration the cells are
% updated as follows:
%  -- If a cell contains a 1 and greater than five of its neighbors
%     contain 1's, then it should become a 0.
%  -- If a cell contains a 0 and has more than three or fewer then
%     three neighbors containing 1's, then it should become a 1
%  -- otherwise the value of a cell remains unchanged.
% The simulation of life iterates Iterations times. The boarder of
% the grid is always zeros. The Grid is Rows-by-Columns in size:
%
% Main(Iterations=4,Rows=10,Columns=10) ->
%
%   Gin :=  0 0 0 0 0 0 0 0 0 0 0 0
%           0 1 1 0 1 0 1 0 0 0 0 0
%           0 1 0 0 1 1 1 0 0 0 0 0
%           0 0 1 0 0 1 1 1 0 1 0 0
%           0 1 1 0 0 0 1 1 1 0 1 0
%           0 1 0 0 0 1 0 1 0 1 0 0
%           0 0 1 0 0 1 0 0 0 1 0 0
%           0 1 1 0 0 1 1 1 1 0 0 0
%           0 1 1 1 0 0 1 0 1 0 0 0
%           0 0 1 0 0 0 0 0 0 0 0 0
```

```
%           0 0 1 1 1 0 1 0 1 0 0 0
%           0 0 0 0 0 0 0 0 0 0 0 0
%
%  ANS :=   0 0 0 0 0 0 0 0 0 0 0 0
%           0 1 1 1 1 1 1 1 1 1 1 0
%           0 1 0 1 1 1 0 0 1 0 1 0
%           0 1 0 1 1 1 0 0 1 1 1 0
%           0 1 0 0 1 1 0 0 1 1 1 0
%           0 1 1 0 0 1 0 0 0 1 1 0
%           0 1 1 1 1 1 0 0 0 0 1 0
%           0 1 1 1 1 0 0 0 0 0 1 0
%           0 1 1 1 1 1 1 1 0 0 1 0
%           0 1 0 1 1 1 1 1 1 0 1 0
%           0 1 1 1 1 1 1 1 1 1 1 0
%           0 0 0 0 0 0 0 0 0 0 0 0
%

type   double  = double_real;
type   OneDim  = array[integer];
type   TwoDim  = array[OneDim];
type     Grid  = array[OneDim];

% PARALLEL RANDOM NUMBER ROUTINES (SEE PREVIOUS EXAMPLE!)
global ranf( Seed:OneDim returns double, OneDim )
global rans( N,Seed1:integer returns TwoDim )

function Convert( Seed:OneDim returns integer, OneDim )
  let
    Number, NewSeed := ranf( Seed );
    V := if Number < 0.5D0 then 1 else 0 end if;
  in
    V, NewSeed
  end let
end function

function Generate( Rows,Columns:integer returns Grid )
  let
    Seed_Stream := rans(Columns,1);
    First := array_fill(0,Columns+1,0);
    Last  := array_fill(0,Columns+1,0);
    Core  := for i in 1,Columns
               Row := for initial
                        j := 1;
                        V,Seed := Convert( Seed_Stream[i] );
                      while ( j < Rows ) repeat
                        j := old j + 1;
                        V,Seed := Convert( old Seed );
                      returns array of V
```

```
                  end for;
             returns array of array_addl(array_addh(Row,0),0)
             end for
  in
    array_addl(array_addh(Core,Last),First)
  end let
end function

function Compute( G:Grid; I,J:integer; returns integer )
  let
    Total := G[I-1,J-1] + G[I-1,J] + G[I-1,J+1] +
             G[I,J-1]        +       G[I,J+1]   +
             G[I+1,J-1] + G[I+1,J] + G[I+1,J+1];
  in
    if ( G[I,J] = 1 & Total > 5 ) then 0
    elseif ( G[I,J] = 0 & Total ~= 3 ) then 1
    else G[I,J] end if
  end let
end function

function DoWork( G:Grid; Rows,Columns:integer returns Grid )
  let
    First := for i in 0,Columns+1 returns array of G[0,i] end for;
    Last  := for i in 0,Columns+1
             returns array of G[Rows+1,i]
             end for;

    Core  := for I in 1, Rows
                Mid := for J in 1, Columns
                       returns array of Compute(G,I,J)
                       end for;
                Row := array_addl(Mid,G[I,0]);
              returns array of array_addh(Row,G[I,Columns+1])
              end for;
  in
    array_addl(array_addh(Core,Last),First)
  end let
end function

function Main( Iterations,Rows,Columns:integer returns Grid,Grid )
  let
    Gin := Generate(Rows,Columns);
  in
    Gin, for initial
           Count := Iterations;
           G := Gin;
         while ( Count > 0 ) repeat
           Count := old Count - 1;
```

```
            G := DoWork( old G, Rows, Columns );
          returns value of G
          end for
   end let
 end function
```

13.15 Particle Transport

```
% *****************************************************************
define Main

%
% Particle-in-Cell Code: Main(n,np,xcell,ycell)
%
%  n is the number of time steps,  np is the number of particles,
%  and xcell and ycell are the number of columns and rows in the
%  grid.
%
% Main(n=1000,np=10,xcell=4,ycell=4) ->
%
%  x :=  8.162e-01 7.828e-01 5.102e-01 2.310e-01 4.547e-01
%        1.521e-01 9.289e-01 7.088e-01 8.280e-01 9.342e-01
%
%  y := 9.004e-01 3.525e-01 3.685e-01 1.799e-02 4.706e-01
%       2.796e-02 4.611e-01 5.679e-01 3.422e-01 3.691e-01
%
%  vx := -4.258e-01 -9.941e-02 -2.799e-01 3.799e-01 2.177e-01
%        -6.611e-01 -1.668e-01  6.246e-02 2.634e-01 5.054e-01
%
%  vy := 4.623e-01 1.947e-01 4.545e-02  4.856e-01 4.705e-01
%        3.897e-02 3.058e-01 2.905e-01 -1.518e-01 3.483e-03
%
% Author: John Feo (Derived From Bmk11A: Los Alamos National Lab.)
%

type double = double_real;
type OneDbl = array[double];
type OneInt = array[integer];

global cos(x: double returns double)
global sin(x: double returns double)
global sqrt(x: double returns double)

function reflect( x,pivot,xmax,delta:double returns double )
  let
    frac := x-double_real(trunc(x/xmax))*xmax
  in
```

```
    if frac = 0.0d0 then pivot-delta else pivot-frac end if
  end let
end function

function move( np,xcell,ycell:integer;
              dt,q,mass,xmax:double;
              ymax,xpcell,ypcell:double;
              xin,yin,vxin,vyin:OneDbl;
              returns OneDbl,OneDbl,OneDbl,OneDbl )
  let
    %
    % Calculate new cell and weight vector
    %
    cell, wght :=

      for i in 1, np

        % grid points numbered from 1
        r_row := yin[i]/ypcell+1.0d0;
        r_col := xin[i]/xpcell+1.0d0;
        row   := trunc(r_row);
        col   := trunc(r_col);

        % percent distance from left and right edge of cell
        lft := r_col-double_real(col);
        rht := 1.0d0-lft;

        % percent distance from top and bottom edge of cell
        bot := r_row-double_real(row);
        top := 1.0d0-bot;

        cell := array [1: row, col];
        wght := array [1: lft, top, rht, bot]

      returns array of cell
              array of wght
      end for;
%
%  Calculate charge density
%
% The charges within a cell are accumulated at the grid points.
% A particle contributes a charge proportional to the area weight
% opposite the grid point. Note, the area weights sum to 1.0, so
% each particle contributes a total charge of q.
%
% Chunk the particles into 10 groups   (we are assuming the number of
% particles is a multiple of ten). Compute a grid of charge densities
% per chunk and then merge the grids.
```

```
%
grids := for i in 1, 10
         returns array of
             for initial
                 k   := (i-1)*np/10;
                 ep  := k+np/10;
                 row := array_fill(1,xcell+1, 0.0d0);
                 rho := array_fill(1,ycell+1, row)
             while k < ep repeat
                 k   := old k+1;
                 r   := cell[k,1];
                 c   := cell[k,2];
                 qsw := q*wght[k,2]*wght[k,3];
                 qse := q*wght[k,1]*wght[k,2];
                 qnw := q*wght[k,3]*wght[k,4];
                 qne := q*wght[k,1]*wght[k,4];
                 rho := old rho[r,c: old rho[r,c] + qsw;
                                r,c + 1: old rho[r,c+1] + qse;
                                r+1,c: old rho[r+1,c] + qnw;
                                r+1,c+1: old rho[r+1,c+1] + qne]
             returns value of rho
             end for
         end for;

  rho := for i in 1,ycell+1 cross j in 1,xcell+1
         returns array of
             for k in 1,10 returns value of sum grids[k,i,j] end for
         end for;
  %
  % Calculate electrostatic potential
  %
  % We use a 5-stencil and Jacoby iteration to solve for the
  % electrostatic potential at (i, j).
  %
  esp := for initial
            i   := 0;
            pi  := 3.1415926d0;
            dx2 := xpcell*xpcell;
            esp := for y in rho at i, j
                   returns array of pi*y*dx2
                   end for
        while i < 10 repeat
            i   := old i + 1;
            esp := for y in old esp at i, j
                       w := if j = 1 then 0.0d0
                            else old esp[i,j-1] end if;
                       n := if i = 1 then 0.0d0
                            else old esp[i-1,j] end if;
```

```
                    e := if j = xcell + 1 then 0.0d0
                         else old esp[i,j+1] end if;
                    s := if i = ycell + 1 then 0.0d0
                         else old esp[i+1,j] end if;
                    z := pi*y*dx2+(w+n+e+s)/4.0d0
                returns array of z
                end for
     returns value of esp
     end for;
%
% Calculate acceleration vectors
%
% The electric field on an edge is the difference of the
% electrostatic potentials at the endpoints divided by the
% distance. At a particle, the electric field in the x (y)
% direction is the weighted sum of the electric field along the
% top and bottom (left and right) edges of the cell. The weight
% is the distance to the opposite edge.
%
ax, ay   := for k in 1, np
              i := cell[k,1];
              j := cell[k,2];
              bot := (esp[i,j]-esp[i,j+1])/xpcell;
              top := (esp[i+1,j]-esp[i+1,j+1]) / xpcell;
              lft := (esp[i,j]-esp[i+1,j])/ypcell;
              rgt := (esp[i,j+1]-esp[i+1,j+1])/ypcell;
              ex := top*wght[k,4]+bot*wght[k,2];
              ey := lft*wght[k,3]+rgt*wght[k,1]
          returns array of ex*q/mass
                  array of ey*q/mass
          end for;
%
% Advance velocity vectors
%
vx1, vy1 := for i in 1, np
            returns array of vxin[i]+ax[i]*dt
                    array of vyin[i]+ay[i]*dt
            end for;
%
% Advance coordinate vectors
%
x1, y1   := for i in 1, np
            returns array of xin[i]+vx1[i]*dt
                    array of yin[i]+vy1[i]*dt
            end for;
%
% Move particles off boundaries and capture escaped particles
%
```

```
    x, vx, y, vy :=
         for i in 1, np
             delta := 0.0000000001d0;
             x, vx := if x1[i] < 0.0d0 then
                         reflect(x1[i],0.0d0,xmax,-delta), -vx1[i]
                      elseif x1[i] = 0.0d0 then
                         x1[i]+delta, -vx1[i]
                      elseif x1[i] > xmax then
                         reflect(x1[i],xmax,xmax,delta), -vx1[i]
                      elseif x1[i] = xmax then
                         x1[i]-delta, -vx1[i]
                      else
                         x1[i], vx1[i]
                      end if;
             y, vy := if y1[i] < 0.0d0 then
                         reflect(y1[i],0.0d0,ymax,-delta), - vy1[i]
                      elseif y1[i] = 0.0d0 then
                         y1[i]+delta, -vy1[i]
                      elseif y1[i] > ymax then
                         reflect(y1[i],ymax,ymax,delta), -vy1[i]
                      elseif y1[i] = ymax then
                         y1[i]-delta, -vy1[i]
                      else
                         y1[i], vy1[i]
                      end if
         returns array of x
                 array of vx
                 array of y
                 array of vy
         end for
  in
    x, y, vx, vy
  end let
end function


function Main( n,np,xcell,ycell:integer
               returns OneDbl, OneDbl, OneDbl, OneDbl )
  for initial

    i  := 0;

    % Assume a space of xmax by ymax
    xmax := 1.0D0;
    ymax := 1.0D0;

    % x and y dimension of each cell
    xpcell := xmax/double_real(xcell);
```

```
  ypcell := ymax/double_real(ycell);

  dt   :=  0.1D0;   % time step
  q    :=  1.0D0;   % charge
  mass :=  1.0D0;   % mass

  % Initialize position vector
  %
  % Distribute the particle in an uniform, counterclockwise spiral
  % from the center of the rectangle.  Let the maximum radius of
  % the spiral be 1/4 of the minor axis of the rectangle.
  %
  pi   := 3.1415926d0;
  xc   := xmax/2.0d0;
  yc   := ymax/2.0d0;
  dmax := min(xmax, ymax);
    rmax := (dmax/8.0d0);
    x, y := for i in 1, np
              r     := rmax*(double_real(i)/double_real(np));
              theta := double_real(i)/(2.0d0*pi)
          returns array of xc+r*cos(theta)
                  array of yc+r*sin(theta)
          end for;

    % Initialize velocity vector
    vx := array_fill(1, np, 0.0D0);
    vy := array_fill(1, np, 0.0D0)

  while i < n repeat

    i := old i + 1;

    x, y, vx, vy :=
          move(np, xcell, ycell, dt, q, mass, xmax, ymax,
               xpcell, ypcell, old x, old y, old vx, old vy)

  returns value of x
          value of y
          value of vx
          value of vy
  end for
end function
```

13.16 Gel Chromatography: SISAL

```
% ***************************************************************
define Main
```

```
%
% See: J. R. Cann and E. York and J. Stewart and J. Vera and R.
%      Maccioni, "Small Zone Gel Chromatography of Interacting
%      Systems: Theoretical and Experimental Evaluation of Elution
%      Profiles for Kinetically Controlled Macromolecule-Ligand
%      Reactions", Analytical Biochemistry, December, 1988, number
%      175.
%
% Author:  J. R. Cann
%
% Main() -> simulates experimentally observed elution patterns of
%           proteins and ligands in a column of gel.
%

type Double = double_real;

type Oned = array[Double];
type Twod = array[Oned];
type Onei = array[integer];
type Twoi = array[Onei];

global LOG( x:Double returns Double )


function RUNKUT(COF1,COF2,COF3,COF4,COF5,COF6,RATIO:Double;
                LN:Twod; N:integer returns Twod )
  let
    rc1,rc2,rc3,rc4,rc5 :=
      for J in 2,N
        CLI      := LN[1,J];
        CMI      := LN[2,J];
        CMLI     := LN[3,J];
        CML2I    := LN[4,J];
        CML2ISOI := LN[5,J];

        RKK1 := COF1*CMI*CLI + COF2*CMLI;
        RKL1 := -(RKK1 + COF3*CMLI*CLI + COF4*CML2I);
        RKP1 := RATIO*(RKK1 + RKK1 + RKL1);
        RKM1 := COF5*CML2I + COF6*CML2ISOI;
        RKN1 := -(RKK1 + RKL1 + RKM1);

        U  := CLI + 0.5D0*RKP1;
        W  := CML2I + 0.5D0*RKN1;
        XX := CMLI + 0.5D0*RKL1;

        RKK2 := COF1*(CMI + 0.5D0*RKK1)*U + COF2*XX;
        RKL2 := -(RKK2 + COF3*XX*U + COF4*W);
```

```
      RKP2 := RATIO*(RKK2 + RKK2 + RKL2);
      RKM2 := COF5*W + COF6*(CML2ISOI + 0.5D0*RKM1);
      RKN2 := -(RKK2 + RKL2 + RKM2);

      VV := CLI + 0.5D0*RKP2;
      Y  := CMLI + 0.5D0*RKL2;
      Z  := CML2I + 0.5D0*RKN2;

      RKK3 := COF1*(CMI + 0.5D0*RKK2)*VV + COF2*Y;
      RKL3 := -(RKK3 + COF3*Y*VV + COF4*Z);
      RKP3 := RATIO*(RKK3 + RKK3 + RKL3);
      RKM3 := COF5*Z +COF6*(CML2ISOI + 0.5D0*RKM2);
      RKN3 := -(RKK3 + RKL3 + RKM3);

      R := CLI + RKP3;
      S := CMLI + RKL3;
      T := CML2I + RKN3;

      RKK4 := COF1*(CMI + RKK3)*R + COF2*S;
      RKL4 := -(RKK4 + COF3*S*R + COF4*T);
      RKP4 := RATIO*(RKK4 + RKK4 + RKL4);
      RKM4 := COF5*T + COF6*(CML2ISOI + RKM3);
      RKN4 := -(RKK4 + RKL4 + RKM4);

      DELK := (RKK1 + RKK2 + RKK2 + RKK3 + RKK3 + RKK4)/6.0D0;
      DELL := (RKL1 + RKL2 + RKL2 + RKL3 + RKL3 + RKL4)/6.0D0;
      DELM := (RKM1 + RKM2 + RKM2 + RKM3 + RKM3 + RKM4)/6.0D0;

      v1 := CLI + RATIO*(DELK + DELK + DELL);
      v2 := CMI + DELK;
      v3 := CMLI + DELL;
      v4 := CML2I - (DELK + DELL + DELM);
      v5 := CML2ISOI + DELM;
    returns array of v1
            array of v2
            array of v3
            array of v4
            array of v5
    end for;

  r1 := array_addl(rc1,0.0D0);
  r2 := array_addl(rc2,0.0D0);
  r3 := array_addl(rc3,0.0D0);
  r4 := array_addl(rc4,0.0D0);
  r5 := array_addl(rc5,0.0D0);
in
  array [1:r1, r2, r3, r4, r5 ]
end let
```

```
end function


function RENUM( NP:integer; LN:Twod; N,I,IELUTE:integer;
                VSEG:Double; CELUTE:Twod returns Twod, Double )
  let
    K := I/IELUTE;
    VOL := double_real(K)*VSEG;

    CELUTE_1 := CELUTE[1,K: LN[1,N];
                       2,K: LN[2,N];
                       3,K: LN[3,N];
                       4,K: LN[4,N];
                       5,K: LN[5,N]  ];
  in
    CELUTE_1, VOL
  end let
end function

function OUT( LN:Twod; N:integer; VOL:Double; KELUTE:integer;
              VSEG,F:Double; CELUTE:Twod
              returns Double, Oned, Oned, Double, Double,
                      Double, Double, Double, Double, Double,
                      integer, Double, Double, Double )
  let
    CTL,
    CTM := for J in 1,KELUTE
             CTL := CELUTE[1,J] + CELUTE[3,J] + CELUTE[4,J] +
                    CELUTE[4,J] + CELUTE[5,J] + CELUTE[5,J];
             CTM := CELUTE[2,J] + CELUTE[3,J] + CELUTE[4,J] +
                    CELUTE[5,J];
           returns array of CTL
                   array of CTM
           end for;

    TOTM,
    TOTML,
    TOTML2,
    TOTML2I := for J in 1,KELUTE
               returns value of sum CELUTE[2,J]
                       value of sum CELUTE[3,J]
                       value of sum CELUTE[4,J]
                       value of sum CELUTE[5,J]
               end for;

    TOT := TOTM + TOTML + TOTML2 + TOTML2I;
    TOTMA := 1.554870369D-05 * 0.486D0;
```

```
    PERML,
    JSTOR,
    STOR,
    PERCENT,
    HL := if (TOT = 0.0D0) then
            0.0D0, 0, 0.0D0, 0.0D0, 0.0D0
          else
            let
              PERML := 100.0D0 * (TOTML + TOTML2 + TOTML2I) / TOT;

              STOR,
              JSTOR := for initial
                         STOR := CTL[915];
                         JSTOR := 915;
                         J := 915;
                       while ( J <= 1190 ) repeat
                         J := old J + 1;

                         STOR,
                         JSTOR := if (old STOR<CTL[old J]) then
                                    old STOR, old JSTOR
                                  else
                                    CTL[old J], old J
                                  end if;
                       returns value of STOR
                               value of JSTOR
                       end for;

              TOT1 := for J in 1,KELUTE
                      returns value of sum CTL[J]
                      end for;

              TOT2 := for J in 1,JSTOR
                      returns value of sum CTL[J]
                      end for;

              PERCENT := 100.0D0*TOT2/TOT1;
              HL :=-LOG(2.0D0)*double_real(JSTOR)*
                   0.016D0/(F*LOG(PERCENT/18.02233D0));
            in
              PERML, JSTOR, STOR, PERCENT, HL
            end let
          end if;
  in
    VOL, CTM, CTL, TOTM, TOTML, TOTML2, TOTML2I, TOT, PERML, TOTMA,
    JSTOR, STOR, PERCENT, HL
  end let
end function
```

```
function FILLUP( N:integer; DX:Double; KELUTE:integer; VSEG:Double;
                 NSEG:integer; GZERO:Oned
                 returns Oned, Twod, Twod, Oned )
  let
    X_c := for J in 2,N
           returns array of double_real(J-1)*DX
           end for;

    X   := array_addl( X_c, 0.0D0 );

    V   := for J in 1,KELUTE
           returns array of double_real(J)*VSEG
           end for;

    C   := for M in 1,5
             Cr := array_fill(2,NSEG,GZERO[M]) ||
                   array_fill(NSEG+1,N,0.0D0);
           returns array of array_addl(Cr,0.0D0)
           end for;

    CELUTE := for I in 1,5 cross J in 1,KELUTE
              returns array of 0.0D0
              end for;
  in
    X, C, CELUTE, V
  end let
end function

function Main( returns Double, Oned, Oned, Double, Double,
                       Double, Double, Double, Double,
                       Double, integer, Double, Double, Double )
  let
    NV := 40000;

    N := 1315;
    NP := N-1;
    NSEG  :=  46;
    KELUTE := 2000;
    IELUTE := 20;
    DX := 0.02D0;

    XI := array[1: 0.8D0, 0.54D0, 0.54D0, 0.54D0, 0.54D0 ];

    RATIO  :=  XI[2] / XI[1];
    DV := XI[1]*DX/double_real(IELUTE);

    AXDIS_1 := array[1: 2.0D-02, 6.0D-02, 6.0D-02, 6.0D-02,6.0D-02 ];
```

```
VEL,
LAMBDA := for M in 1,5
            VEL_1  := (1.0D0/XI[M])-(1.0D0/XI[1]);
            AXDIS  := AXDIS_1[M]-0.5D0*VEL_1*DX;
            VEL    := DV*VEL_1/DX;
            LAMBDA := DV*AXDIS/(exp(DX,2.0D0));
          returns array of VEL
                  array of LAMBDA
          end for;


VSEG := XI[1]*DX;
F  :=  20.6D0 / 3600.0D0;
EK0  :=  1.1D05;
RDML  :=  0.5D0;
RAML  :=  2.0D0*EK0*RDML;
EKISOM  :=  20.0D0;
EKAPP  :=  0.5D0*EK0/(1.0D0 + EKISOM);
RDML2  :=  1.0D0;
RAML2  :=  EKAPP*RDML2;
RRISOM  :=  1.0D-03;
RFISOM  :=  EKISOM*RRISOM;
COF1  :=  -RAML * DV / F;
COF2  :=  RDML * DV / F;
COF3  :=  RAML2 * DV / F;
COF4  :=  -RDML2 * DV / F;
COF5  :=  RFISOM * DV / F;
COF6  :=  -RRISOM * DV / F;

GZERO := array[1: 0.4038637706D-05, 0.7454342552D-05,
                  0.6623185565D-05, 0.0070055980D-05,
                  0.1401119600D-05 ];

VOL,
CELUTE,
LN := for initial
        VOL := 0.0D0;
        I := 1; S := 0;
        X, LN, CELUTE, V := FILLUP(N,DX,KELUTE,VSEG,NSEG,GZERO);

      while ( I <= NV ) repeat
        I := old I + 1;

        LN1 :=
          for M in 1,5
            Sp2 := old S+2; Sp3 := old S+3; SpN := old S+N;
```

```
                LNrc := for JJ in 3, NP
                          J := JJ + old S;
                          V :=
                            old LN[M,J]+LAMBDA[M]*(old LN[M,J+1]-
                            old LN[M,J]-old LN[M,J]+old LN[M,J-1])-
                            VEL[M]*(old LN[M,J]-old LN[M,J-1]);
                        returns array of V
                        end for;

                LNc2 := old LN[M,Sp2]+LAMBDA[M]*
                        (old LN[M,Sp3]-old LN[M,Sp2])-
                        VEL[M]*old LN[M,Sp2];

                LNn  := old LN[M,SpN]+LAMBDA[M]*
                        (old LN[M,SpN-1]-old LN[M,SpN]);

                LNr  := array_addl(array_addh(LNrc,LNn),LNc2);
              returns array of array_addl(LNr,0.0D0)
              end for;

            LN := RUNKUT(COF1,COF2,COF3,COF4,COF5,COF6,RATIO,LN1,N);

            S, CELUTE,
            VOL := if ((old I/IELUTE)*IELUTE = old I) then
                     -1,
                     RENUM(NP,LN,N,old I,IELUTE,VSEG,old CELUTE)
                   else
                     0, old CELUTE, old VOL
                   end if;
          returns value of VOL
                  value of CELUTE
                  value of LN
          end for;
  in
    OUT(LN,N,VOL,KELUTE,VSEG,F,CELUTE)
  end let
end function
```

13.17 Gel Chromatography: Fortran

```
C ****************************************************************
      PROGRAM RICARDO12A

C
C See: J. R. Cann and E. York and J. Stewart and J. Vera and R.
C      Maccioni, "Small Zone Gel Chromatography of Interacting
C      Systems: Theoretical and Experimental Evaluation of Elution
```

```
C      Profiles for Kinetically Controlled Macromolecule-Ligand
C      Reactions", Analytical Biochemistry, December, 1988, number
C      175.
C
C Author:  J. R. Cann
C
C Main() -> simulates experimentally observed elution patterns of
C           proteins and ligands in a column of gel.
C

      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON CLN(1315,5),CL(1315,5),X(1315),CELUTE(5,2000),V(2000),
     +GZERO(5)
      DIMENSION VEL(5),AXDIS(5),XI(5)
      REAL*8  LAMBDA(5),t1,t2

      NV=40000
      IOUT=40000
      N=1315
      NP=N-1
      NSEG = 46
      KELUTE=2000
      IELUTE=20
      DX=0.02
      XI(1)=0.8
      XI(2)=0.54
      XI(3)=0.54
      XI(4)=0.54
      XI(5)=0.54
      RATIO = XI(2) / XI(1)
      DV=XI(1)*DX/DBLE(IELUTE)
      AXDIS(1)=2.0E-02
      AXDIS(2)=6.0E-02
      AXDIS(3)=6.0E-02
      AXDIS(4)=6.0E-02
      AXDIS(5)=6.0E-02
      DO 1 M = 1,5
        VEL(M)=(1.0/XI(M))-(1.0/XI(1))
        AXDIS(M)=AXDIS(M)-0.5*VEL(M)*DX
        VEL(M)=DV*VEL(M)/DX
        LAMBDA(M)=DV*AXDIS(M)/(DX**2)
  1   CONTINUE
      VSEG=XI(1)*DX
      F = 20.6 / 3600.0
      EKO = 1.1E05
      RDML = 0.5
      RAML = 2.0*EKO*RDML
      EKISOM = 20.0
```

```
      EKAPP = 0.5*EK0/(1.0 + EKISOM)
      RDML2 = 1.0
      RAML2 = EKAPP*RDML2
      RRISOM = 1.0E-03
      RFISOM = EKISOM*RRISOM
      COF1 = -RAML * DV / F
      COF2 = RDML * DV / F
      COF3 = RAML2 * DV / F
      COF4 = -RDML2 * DV / F
      COF5 = RFISOM * DV / F
      COF6 = -RRISOM * DV / F
      GZERO(1)=0.4038637706E-05
      GZERO(2)=0.7454342552E-05
      GZERO(3)=0.6623185565E-05
      GZERO(4)=0.0070055980E-05
      GZERO(5)=0.1401119600E-05
      L=1
      LN=2
      VOL=0.0
      CALL FILLUP(N,DX,L,LN,KELUTE,VSEG,NSEG)

      DO 2 I = 1,NV

        DO 3 M = 1,5
          CLN(2,M)=CL(2,M)+LAMBDA(M)*(CL(3,M)-CL(2,M))-VEL(M)*CL(2,M)
3       CONTINUE

        DO 4 M=1,5
          DO 6 J=3,NP
            CLN(J,M)=CL(J,M)+LAMBDA(M)*(CL(J+1,M)-CL(J,M)
     X      -CL(J,M)+CL(J-1,M))-VEL(M)*(CL(J,M)-CL(J-1,M))
6         CONTINUE
          CLN(N,M)=CL(N,M)+LAMBDA(M)*(CL(N-1,M)-CL(N,M))
4       CONTINUE

        CALL RUNKUT (COF1,COF2,COF3,COF4,COF5,COF6,RATIO,LN,N)

        IF ((I/IELUTE)*IELUTE.EQ.I)
     X     CALL RENUM(NP,LN,N,I,IELUTE,VOL,VSEG)

2     CONTINUE

      CALL OUT(L,N,VOL,KELUTE,VSEG,F)
      STOP
      END

      SUBROUTINE FILLUP(N,DX,L,LN,KELUTE,VSEG,NSEG)
      IMPLICIT REAL*8 (A-H,O-Z)
```

```
      COMMON CLN(1315,5),CL(1315,5),X(1315),CELUTE(5,2000),V(2000),
     +GZERO(5)
      X(1)=0.0
      DO 1 J = 2,N
      X(J)=DBLE(J-1)*DX
1     CONTINUE
      DO 2 M = 1,5
        CL(1,M)=0.0
        CLN(1,M)=0.0
        DO 3 J = 2,NSEG
          CL(J,M)=GZERO(M)
          CLN(J,M)=GZERO(M)
3       CONTINUE
          DO 4 J = 1,KELUTE
            CELUTE(M,J)=0.0
4         CONTINUE
2     CONTINUE
      DO 5 J = 1,KELUTE
        V(J)=DBLE(J)*VSEG
5     CONTINUE
      RETURN
      END

      SUBROUTINE RENUM(NP,LN,N,I,IELUTE,VOL,VSEG)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON CLN(1315,5),CL(1315,5),X(1315),CELUTE(5,2000),V(2000),
     +GZERO(5)
      K=I/IELUTE
      VOL=DBLE(K)*VSEG
      NPP = NP - 1
      MN = N + 1

      DO 1 M = 1,5
      CELUTE(M,K)=CL(N,M)

      DO 2 J = 1,NPP
        JM = MN-J
        CL(JM,M)=CL(JM-1,M)
2     CONTINUE

      CL(2,M)=0.0
1     CONTINUE
      RETURN
      END

      SUBROUTINE RUNKUT (COF1,COF2,COF3,COF4,COF5,COF6,RATIO,LN,N)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON CLN(1315,5),CL(1315,5),X(1315),CELUTE(5,2000),V(2000),
```

```
    +GZERO(5)
     DO 1 J = 2,N
       CMI = CLN(J,2)
       CLI = CLN(J,1)
       CMLI = CLN(J,3)
       CML2I = CLN(J,4)
       CML2ISOI = CLN(J,5)
       RKK1 = COF1*CMI*CLI + COF2*CMLI
       RKL1 = -(RKK1 + COF3*CMLI*CLI + COF4*CML2I)
       RKP1 = RATIO*(RKK1 + RKK1 + RKL1)
       RKM1 = COF5*CML2I + COF6*CML2ISOI
       RKN1 = -(RKK1 + RKL1 + RKM1)
       U = CLI + 0.5*RKP1
       W = CML2I + 0.5*RKN1
       XX = CMLI + 0.5*RKL1
       RKK2 = COF1*(CMI + 0.5*RKK1)*U + COF2*XX
       RKL2 = -(RKK2 + COF3*XX*U + COF4*W)
       RKP2 = RATIO*(RKK2 + RKK2 + RKL2)
       RKM2 = COF5*W + COF6*(CML2ISOI + 0.5*RKM1)
       RKN2 = -(RKK2 + RKL2 + RKM2)
       VV = CLI + 0.5*RKP2
       Y = CMLI + 0.5*RKL2
       Z = CML2I + 0.5*RKN2
       RKK3 = COF1*(CMI + 0.5*RKK2)*VV + COF2*Y
       RKL3 = -(RKK3 + COF3*Y*VV + COF4*Z)
       RKP3 = RATIO*(RKK3 + RKK3 + RKL3)
       RKM3 = COF5*Z +COF6*(CML2ISOI + 0.5*RKM2)
       RKN3 = -(RKK3 + RKL3 + RKM3)
       R = CLI + RKP3
       S = CMLI + RKL3
       T = CML2I + RKN3
       RKK4 = COF1*(CMI + RKK3)*R + COF2*S
       RKL4 = -(RKK4 + COF3*S*R + COF4*T)
       RKP4 = RATIO*(RKK4 + RKK4 + RKL4)
       RKM4 = COF5*T + COF6*(CML2ISOI + RKM3)
       RKN4 = -(RKK4 + RKL4 + RKM4)
       DELK = (RKK1 + RKK2 + RKK2 + RKK3 + RKK3 + RKK4)/6.0
       DELL = (RKL1 + RKL2 + RKL2 + RKL3 + RKL3 + RKL4)/6.0
       DELM = (RKM1 + RKM2 + RKM2 + RKM3 + RKM3 + RKM4)/6.0
       CL(J,2) = CMI + DELK
       CL(J,3) = CMLI + DELL
       CL(J,1) = CLI + RATIO*(DELK + DELK + DELL)
       CL(J,5) = CML2ISOI + DELM
       CL(J,4) = CML2I - (DELK + DELL + DELM)
1    CONTINUE
     RETURN
     END
```

```
      SUBROUTINE OUT(L,N,VOL,KELUTE,VSEG,F)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON CLN(1315,5),CL(1315,5),X(1315),CELUTE(5,2000),V(2000),
     +GZERO(5)
      DIMENSION CTL(2000),CTM(2000)

      DO 2 J = 1,KELUTE
        CTL(J) = CELUTE(1,J) + CELUTE(3,J) + CELUTE(4,J) +
     X           CELUTE(4,J) + CELUTE(5,J) + CELUTE(5,J)
        CTM(J) = CELUTE(2,J) + CELUTE(3,J) + CELUTE(4,J) +
     X           CELUTE(5,J)
 2    CONTINUE

      TOTM = 0.0
      TOTML = 0.0
      TOTML2 = 0.0
      TOTML2I = 0.0
      DO 4 J = 1,KELUTE
        TOTM = TOTM + CELUTE(2,J)
        TOTML = TOTML + CELUTE(3,J)
        TOTML2 = TOTML2 + CELUTE(4,J)
        TOTML2I = TOTML2I + CELUTE(5,J)
 4    CONTINUE

      TOT = TOTM + TOTML + TOTML2 + TOTML2I

      IF (TOT .EQ. 0.0) THEN
        GO TO 6
      ENDIF

      PERML = 100.0 * (TOTML + TOTML2 + TOTML2I) / TOT

      TOTMA = 1.554870369E-05 * 0.486

      TOT1=0.0
      TOT2=0.0
      STOR=CTL(915)
      JSTOR=915
      DO 7 J=1,KELUTE
        TOT1=TOT1 + CTL(J)
        IF (J.LT.915.OR.J.GT.1190) GO TO 7
        IF (STOR.LT.CTL(J)) GO TO 7
        STOR=CTL(J)
        JSTOR=J
 7    CONTINUE
      DO 8 J=1,JSTOR
        TOT2=TOT2 + CTL(J)
 8    CONTINUE
```

```
      PERCENT=100.0*TOT2/TOT1
      HL=-LOG(2.0)*DBLE(JSTOR)*0.016/(F*LOG(PERCENT/18.02233))

      WRITE (0,100) VOL
100   FORMAT (' ',' ',2X,'VOLUME OF ELUATE =',E12.6,2X,'ML')
      WRITE (0,200)
200   FORMAT(' ',' ',9X,'POSITION',9X,'CONCL',10X,'CONCM',8X,
     X'CONCML',9X,'CONCML2',9X,'CONCML2*')
      DO 1 J=1,60
        WRITE (0,300) X(J),C(1,J,L),C(2,J,L),C(3,J,L),C(4,J,L),
     X                C(5,J,L)
  1   CONTINUE
300   FORMAT(' ',2X,6E15.6)
      WRITE (0,400)
400   FORMAT (' ',' ',130('_'))
      WRITE (0,500)
500   FORMAT (' ',' ',9X,'VOLUME',11X,'CONCL',9X,'CONCM',10X,
     X        'CONCML',
     X8X,'CONCML2',8X,'CONCML2*',7X,'TOTAL M',8X,'TOTAL L')
      DO 3 J = 1,KELUTE,4
        WRITE (0,600) V(J),CELUTE(1,J),CELUTE(2,J),CELUTE(3,J),
    X                 CELUTE(4,J),CELUTE(5,J),CTM(J),CTL(J)

        WRITE(0,612)

        DO 3 J = 1,KELUTE,4
          WRITE (0,600) CTM(J), CTM(J+1), CTM(J+2), CTM(J+3)
  3   CONTINUE

      WRITE (0,610)

      DO 40 J = 1,KELUTE,4
        WRITE (0,600) CTL(J), CTL(J+1), CTL(J+2), CTL(J+3)
 40   CONTINUE

600   FORMAT (' ',4(2X,1E15.6))
610   FORMAT (' ',2X,' CTL ')
612   FORMAT (' ',2X,' CTM ')

      WRITE (0,400)

      WRITE (0,700) TOTM,TOTML
      WRITE (0,701) TOTML2,TOTML2I
      WRITE (0,750) TOT
      WRITE (0,800) PERML
700   FORMAT (' ',' ',2X,'TOTAL M =',E12.6,2X,'TOTAL ML =',E12.6,
     X2X)
701   FORMAT (' ',' TOTAL ML2 =',E12.6,2X,'TOTAL  ML2I =',E12.6)
```

```
750   FORMAT (' ',' ',2X,'TOTAL (M + ML + ML2 + ML2*) =',E12.6)
800   FORMAT (' ',' ',2X,'PERCENTAGE (ML + ML2 + ML2*) = ',F6.2)
      WRITE (0,900) TOTMA
900   FORMAT (' ',' ',2X,
     X'TOTAL (M + ML + ML2 + ML2*) APPLIED TO COLUMN = ',E12.6)
      WRITE (0,400)

      WRITE (0,910) JSTOR,STOR
910   FORMAT (' ',' ',2X,'MINIMUM J =',I4,2X,
     X'MINIMUM LIG CONC =',E12.6)
      WRITE (0,920) PERCENT,HL
920   FORMAT (' ',' ','PERCENT FAST PEAK =',F9.5
     X,2X,'APPARENT HALF-LIFE =',F11.4)
      WRITE (0,400)
  6   RETURN
      END
```

Acknowledgements

I would like to thank Tom Deboni for reviewing and improving this document.

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] Mathew Burns. Sisal challenges Fortran. *Supercomputing Review*, 5(2):72–73, February 1992.

[2] D. C. Cann. Retire Fortran? A Debate Rekindled. In *Supercomputing 91*, November 1991.

[3] D. C. Cann. The optimizing SISAL compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.

[4] J. R. Cann, E. York, J. Stewart, J. Vera, and R. Maccioni. Small zone gel chromatography of interacting systems: Theoretical and experimental evaluation of elution profiles for kinetically controlled macromolecule-ligand reactions. *Analytical Biochemistry*, (175), December 1988.

[5] W. P. Crowley, C. P. Henderson, and T. E. Rudy. The simple code. Technical Report UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, Februrary 1978.

[6] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[7] S. K. Skedzielewski and R. Kim Yates. Fibre: An external format for SISAL and IF1 data objects, version 1.0. Manual M-154, Lawrence Livermore National Laboratory, Livermore, CA, January 1985.

[8] Lisa Stapleton. Livermore labs offers cray time—free. *Supercomputing Review*, 4(5):14–15, May 1991.

Disclaimer