219 942

UCRL-JC-110915
PREPRINT

Sisal

John T. Feo
Lawrence Livermore National Laboratory
Livermore, California

This paper was prepared as a book chapter for
A Comparative Study of Parallel Programming Languages: The Salishan Problems

July 1992

Sisal

John Feo
Computing Research Group, L-306
Lawrence Livermore National Laboratory
Livermore, CA 94550

1. Introduction

Sisal, **S**treams and **I**terations in a **S**ingle **A**ssignment **L**anguage, a derivative of Val, was defined in 1983 [4] and revised in 1985 [5]. Since 1985 the language definition has remained constant providing a stable testbed for programming language research and functional program development. The Sisal Language Project began as a collaborative effort between Lawrence Livermore National Laboratory, Colorado State University, University of Manchester, and Digital Equipment Corporation. Today, only LLNL and CSU continue to develop and promote the language; however, Sisal and its intermediate form IF1 [7] are being used by research groups across the United States and around the World.

The Project has six objectives:

1. to define a general-purpose functional language,
2. to define a language-independent intermediate form for dataflow graphs,
3. to develop optimization techniques for high performance parallel applicative computing,
4. to develop a microtasking environment that supports dataflow on conventional computer systems,

5. to achieve execution performance comparable to imperative languages, and
6. to validate the functional style of programming for large-scale scientific applications.

The objectives emphasize usability and performance. Goals four, five, and six set the Sisal effort apart from other functional language projects. They reflect the computing environment at Lawrence Livermore National Laboratory and other government facilities.

Functional languages promote the development of correct, determinate parallel programs. Functional programs are free of aliases, side effects, and time dependent errors. Results are determinate regardless of architecture, operating system, or execution environment. Unlike parallel imperative languages, functional languages decrease the programming burden. Users can define only what is to be computed and can encode only the data dependencies among operations. The compiler and the run time system are responsible for scheduling operations, communicating data values, synchronizing operations, and managing memory. It is as easy to write a functional program, which is implicitly parallel, as it is to write a sequential imperative program. Relieved of parallel programming's most difficult chores, the user is free to concentrate on algorithm design and application development.

Section 2 introduces Sisal, discussing those language features used in subsequent sections. For the full language definition see [5]. In Sections 3, 4, 5, and 6 we present solutions to the four Salishan problems. Since functional programming requires a different thought process than imperative programming, we describe how we formulated each solution. In Section 7, we conclude with some general remarks regarding Sisal and functional programming.

2. Language Definition

Sisal is a strongly typed, general purpose functional language that supports data types and operations for scientific computing. To minimize learning time and enhance readability, the creators of Sisal adopted a Pascal-like, block syntax delimited by keywords. Since most scientific programs are written once but maintained over many years, the designers sought to improve readability wherever possible. Sisal programs are slightly wordy, but easy to read and to understand. Some language features exchange elegance for readibility.

Sisal has several important semantic properties. First, the language is mathematically sound—functions map inputs to outputs without side effects. Second, names are referentially transparent; that is, they stand for values rather than memory locations. Third, the language is single-assignment. A name may be assigned a value only once within each scope. A Sisal tenet, not required by its functional semantics but enforced by the compiler to aid readability, is that all names must be defined before they are used.

2.1. Types

Sisal supports the standard scalar data types: **boolean**, **char**, **integer**, **real**, and **double_real**. It also includes the aggregates: **array**, **record**, **stream**, and **union**. Users may define aggregate types by using the **type** statement. For example,

```
type Istr = stream [integer];
type OneR = array  [real];
type TwoR = array  [OneR];
```

define, respectively, a stream of integers, an array of reals, and an array of arrays of reals. The latter is equivalent mathematically to a two-dimensional array of real values.

In Sisal both arrays and streams are homogeneous aggregates of any standard or user-defined type. Arrays support random access whereas stream elements are available only in FIFO order. That is the i-th element of a stream must be consumed before the $(i + 1)$-st element may be consumed. Consequently, Sisal streams cannot deadlock. Array declarations include neither size nor bounds information. An array's size, lower bound, and shape are determined during execution. Since the components of a multi-dimensional array are arrays, each may have a different length and lower bound. We say that Sisal's arrays are *ragged*.

The types of names are not declared; instead, the compiler infers the type of each name from the surrounding context. The two exceptions are the formal parameters and results of functions. These too could be inferred, but at the expense of readability. The types are declared in the function headers.

2.2. Functions

A function can take zero or more arguments and must return one or more values. The type of each formal parameter and result value is declared in the function header. For example, the function `circle`

```
function circle(radius: real returns real, real)
  2.0 * 3.14 * radius, 3.14 * radius * radius
end function
```

takes one formal parameters `radius`, of type **`real`**, and returns two **`real`** values, the circumference and area of the circle. The number of values a function or an expression returns is referred to as the *arity* of the function or expression. A function has access to only its arguments. There are no global values and functions do not retain state between invocations. The effect of invoking a function is limited to the values it returns—there are no side effects.

A function name and list of actual parameters can appear anywhere an expression of the same type and arity can appear. For example, the statements

```
    circum, area := circle(radius);
```

and

```
    a, b := new_function(circle(radius));
```

are legal statements provided `new_function` takes two real values as input and returns two results. But the statement

```
    circum := circle(radius)
```

is illegal because there is only a single name on the left-hand side.

2.3. Let expressions

The **`let`** expression defines a set of names, and then uses the names to compute one or more values. The expression,

```
    let
      ...          % the let clause
    in
      ...          % the in clause
    end let
```

consists of two clauses: a **let** clause and an **in** clause. The latter defines the arity, type, and value of the **`let`** expression. For example, the expression

```
    let
      pi := 3.14
    in
      2.0 * pi * radius, pi * radius * radius
    end let;
```

has arity two, type (**real**, **real**), and returns the circumference and area of a circle. The **let** expression is equivalent to the function `circle` defined before and could replace the two expressions in the function's body.

2.4. For expressions

The **for** expression,

```
for <range generator>
  <loop body>
returns <returns clause>
end for
```

is the parallel loop form in Sisal. It has three parts: a **range generator**, a **loop body**, and a **returns** clause.

The **range generator** is a dot or cross product of a set of sequences or scatters (see Figure 1). An instance of the loop body is executed for each index, value, or *n*-tuple of the range. The range generator specifies the order of reduction, and defines the size and structure of any generated aggregate object. For example, the expression

```
for i in 1,n cross j in 1,m
returns array of (i + j)
end for
```

returns a two dimensional array of *n* rows and *m* columns. At first, many Sisal programmers fail to understand the subtleties of this syntax. A common mistake is to write the transpose of an (*n* x *m*) matrix as

```
for i in 1,n cross j in 1,m
returns array of X[j, i]
end for
```

Range Generator	Comments
`for x in A`	a scatter
`for i in 1, n`	a sequence
`for x in A dot y in B`	a dot product of two scatters
`for i in 1, n cross j in 1, n`	a cross product of two sequences

Figure 1 - Forms of the Range Generator

But this returns an *(n* x *m)* matrix and not an *(m* x *n)* matrix. The correct expression is

```
for i in 1,m cross j in 1,n
returns array of X[j, i]
end for
```

The **loop body** is a set of name definitions. Sisal's semantics prevents an instance of the loop body from referencing values computed by any other instance. Thus, the instances of the loop body are data independent and may be executed in parallel.

The **returns** clause defines the arity, type, and value of the `for` expression. Each result is a reduction of values defined in the loop body. The order of reduction is determinate and equivalent to the sequential execution of the loop bodies. Figure 2 lists the eight reduction operations supported by Sisal. The values contributing to a reduction can be filtered by including a `when` clause. For example,

```
array of x when x > 0
```

returns an array of only positive values. If none of the values are positive, an empty array of the same type as `x` is returned

As an illustration of the `for` expression, consider matrix multiplication. Let *a* and *b* be real arrays of size *(n* x *m)* and *(m* x *n)*, respectively, then the product of *a* and *b* is

Returns Clause	Comments
`value of x`	returns the last value of x
`array of x`	returns an array of x values
`stream of x`	returns a stream of x values
`value of sum x`	returns the sum of x values
`value of product x`	returns the product of x values
`value of least x`	returns the smallest x value
`value of greatest x`	returns the largest x value
`value of catenate x`	returns the array formed by catenating the x values (note x must be an array)

Figure 2 - The Returns Clause

$$\sum_{k=1}^{m} a(i,\ k) \bullet b(k, j), \qquad 1 \le i, j \le n \qquad (2.1)$$

We begin by writing

```
function mmul(n,m: integer; a,b: TwoR returns TwoR)
   ...
end function
```

which defines `mmul` as a function of four inputs: `n` and `m` of type **integer**, and `a` and `b` of type `TwoR` (defined previously). The function returns a single value of type `TwoR`, the product of *a* and b.

The function's body is the Sisal expression for Equation 2.1. Since the equation computes n^2 independent real values and assembles them into an array, we want to use the following **for** expression

```
for i in 1, n cross j in 1, n
  Cij :=
returns array of Cij
end for
```

`Cij` is the inner product of the *i*-th row of *a* and *j*-th column of *b*. We write the inner product of two *m*-element vectors as

```
for k in 1, m
returns value of sum a[i,k] * b[k,j]
end for
```

Putting everything together, we have

```
function mmul(n,m: integer; a,b: TwoR returns TwoR)
   for i in 1, n cross j in 1, n
     Cij := for k in 1, m
             returns value of sum a[i,k] * b[k,j]
             end for
   returns array of Cij
   end for
end function
```

There are three important observations to make. First, the function expresses only the necessary mathematical computations. The implementation of the function, which is 100% parallel, is unspecified. Second, the function is dynamic. The resources necessary to execute the expression depend entirely on *n* and *m*. There is no static allocation of memory or processors. Third, unlike an imperative function which would first allocate the memory for the result and then fill it in, the Sisal function consists of a single expression that both creates and defines the result.

2.5. For Initial expressions

The **`for initial`** expression,

```
for initial                          for  initial
   <initialization>                     <initialization>
while <test> repeat                  repeat
   <loop body>                          <loop body>
returns <returns clause>             until <test>
end  for                             returns <returns clause>
                                     end  for
```

permits loop carried dependencies, but retains single assignment semantics. It comprises four segments: **initialization**, **test**, **loop body**, and **result** clause. The **initialization** segment defines all loop constants and assigns initial values to all loop-carried names. It is the first iteration of the loop.

The **test** may appear either before or after the body. Two forms are supported in Sisal: **while** *<test>* and **until** *<test>*. The **loop body** computes new values for all loop-carried names. An instance of the body may refer to any loop-carried name defined in the previous instance by prefixing the name with the keyword **old**. Thus, **old** a refers to the value of a on the previous iteration. Note that defining the value of a on the present iteration does not destroy the old value. The rebinding of loop-carried names to values is implicit and occurs between iterations.

The **returns** clause of the **for initial** expression is identical in syntax and semantics to the returns clause of the **for** expression.

As an example of the **for initial** expression, consider the function first_sum. Let x be real vector of length n, then $y(i)$ is

$$y(i) = \sum_{j=1}^{i} x(j), \qquad 1 \le i \le n \qquad (2.2)$$

While the elements of y are data independent, using a **for** expression would result in $O(n^2)$ additions. A more efficient algorithm can be had using a **for initial** expression.

First, we rewrite Equation 2 as

$$y(i) = \begin{cases} x(1) & i = 1 \\ y(i - 1) + x(i) & 2 \leq i \leq n \end{cases} \qquad (2.3)$$

and then write Equation 2.3 in Sisal

```
function first_sum(n: integer; x: OneR returns OneR)
  for initial
    i := 1;
    y := x[1]
  while i < n repeat
    i := old i + 1;
    y := old y + x[i]
  returns array of y
  end for
end function
```

Again, the translation from mathematics to Sisal is straightforward. The *how* of the loop—allocating memory for the result and assigning values to positions—is implicit, implied by the semantics of the expression.

2.6. Array and stream operations

Figures 3 and 4 list some of the array and stream operations in Sisal. It is important to remember that these, and all operations in Sisal, retain single-assignment semantics. The statement

```
A[1: 0, 1, 2]
```

does not replace the first, second, and third elements of *A*, as in an imperative language, but instead, creates a new array identical to *A* with the first, second, and third elements set to 0, 1, and 2. The function

```
stream_first(A)
```

Array Operations	Comments
`array OneR`	creates an empty array (type is required)
`array OneR [1: 1.0, 2.0,`	creates an array of real values with lower bound 1 (type is optional)
`array_fill(1, n, 0)`	creates an array of 0s, lower bound 1, upper bound n
`A[1: 0, 1, 2]`	creates a new array idential to A with elements 1, 2, and 3 set to 0, 1, and 2
`A || B`	creates a new array which is the catena-tion of A and B

Figure 3 - Array Operations

Stream Operations	Comments
`stream StrI [1, 2, 3,`	creates a stream of integer values (type is optional)
`stream_append(A, v)`	creates a new stream idential to A with the element v appended to the tail
`A || B`	creates a new stream which is the cate-nation of A and B
`stream_rest(A)`	creates a new stream identical to A with the first element removed
`stream_first(A)`	returns the first element of A
`stream_empty(A)`	true if A is empty and the producer has terminated

Figure 4 - Stream Operations

returns the first element of *A* but does not modify *A* (i.e., it does not remove the first element). To define a stream comprised of all the elements of *A* except the first element, the user must write

```
B := stream_rest(A)
```

`stream_empty(A)` returns true only if all of `A` is consumed and the producer of `A` has terminated. It does not return true simply because no value is available. Such behavior would introduce non-determinism and is not supported. Moreover, `stream_first` always returns a value. If the producer is slow and no value is available, the function waits for a value. If the stream is empty and the producer has terminated, an error value is returned.

On close inspection of the array operations listed in Figure 3 and the loop structures described in the previous section one might conclude that Sisal's single-assignment semantics, and the copy and memory management operations implied by those semantics, would make Sisal inappropriate for scientific computing. However, studies show [1, 6] that compile-time analysis can eliminate virtually all unnecessary copy and memory management operations. In fact, Sisal's conservative semantics and explicit array operations aid the analysis. We fully expect to *achieve execution performance comparable to imperative languages.*

3.0. Hamming's Problem, Extended

3.1. Understanding Hamming's Problem

Given an integer n and a set of primes $\{a, b, c, \ldots\}$, generate in order and without duplicates all integers of the form

$$a^i \; b^j \; c^k \; \ldots \leq n$$

One way to solve Hamming's Problem is to compute the cross product of the sets

$$\{a^i \mid i \geq 0\},\ \{b^j \mid j \geq 0\},\ \{c^k \mid k \geq 0\},\ \ldots$$

forming a set of tuples of the form

$$(a^i,\ b^j,\ c^k,\ \ldots) \qquad i,\ j,\ k \geq 0$$

Multiplying together the elements of each tuple, sorting the products in increasing order, and discarding the integers greater than n, solves Hamming's Problem.

For example, let $n = 10$ and the set of primes be {2, 3, 5}. First we form the three sets

$$\{1, 2, 4, 8\},\ \{1, 3, 9\},\ \{1, 5\}$$

Note that we have discarded integers greater than 10. Next we form the cross product of the three sets

```
{ (1,1,1), (1,1,5), (1,3,1), (1,3,5), (1,9,1), (1,9,5), (2,1,1), (2,1,5),
  (2,3,1), (2,3,5), (2,9,1), (2,9,5), (4,1,1), (4,1,5), (4,3,1), (4,3,5),
  (4,9,1), (4,9,5), (8,1,1), (8,1,5), (8,3,1), (8,3,5), (8,9,1), (8,9,5)
}
```

Multiplying together the elements of each tuple, sorting the products in increasing order, and discarding values greater than 10, yields the set

$$\{ 1, 2, 3, 4, 5, 6, 8, 9, 10 \}$$

Figure 5a shows a task graph to compute the powers of 2. It consists of a task and three edges: *i*, *s*, and *b*. The edges act as FIFO queues of zero or more values. The instruction set for the task is:

```
1. remove the token on edge i, call its value x
2. repeat
    3. output x on edge s
    4. output 2 times x on edge b
    5. remove the token on edge b, call its value x
```

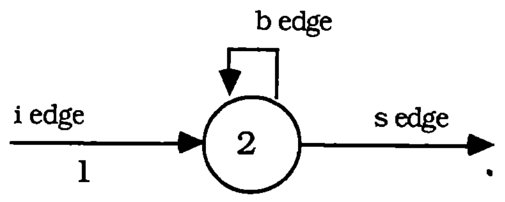


Figure 5a - Task graph to compute the powers of 2

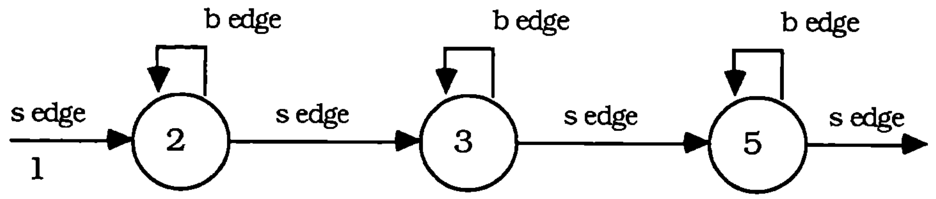


Figure 5b - Task graph to compute $2^i\ 3^j\ 5^k$

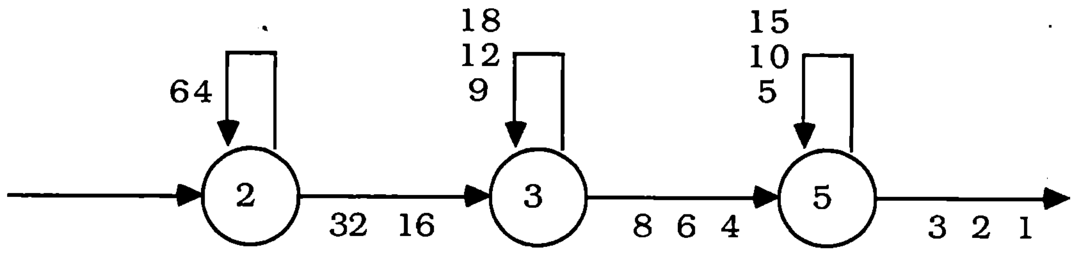


Figure 5c - A snapshot of task graph 5b

Figure 5b shows three instances of the task graph wired together. The tokens issued on the rightmost *s* edge are the solution to Hamming's Problem for the set of primes {2, 3, 5}. The instruction set for the second and third task does require a slight modification. On each iteration, except the first, the tasks have a choice of removing a token from either of the incoming *s* or *b* edges. By removing the smaller of the two values, the tasks order the values output on the outgoing *s* edge. Figure 5c shows a snapshot of task graph in Figure 5b.

3.2. A Sisal solution

In this section we explain how to express the task graph shown in Figure 5b in Sisal. We use a **for initial** expression and streams to spawn a task per prime and establish a FIFO queue between tasks *i* and *i* + 1, *i* ≥ 1. The Sisal code is

```
for initial
   i := 0;
   s_stream := stream[1]
while i < array_size(primes) repeat
   i := old i + 1;
   s_stream := powers(n, primes[i], old s_stream)
returns value of s_stream
end for
```

The initialization clause defines `i` and `s_stream`, an integer stream of one element. The loop body is executed iteratively, once for each prime. The body increments `i` and calls the function `powers` which takes three arguments: *n*, the *i*-th prime, and **old** `s_stream` (the value of `s_stream` on the previous iteration). The value of the **for initial** expression is the value of `s_stream` defined on the last iteration (i.e., returned by the last call to `powers`). The **for initial** expression invokes an instance of `powers` for each prime and establishes the FIFO queue `s_stream` between successive instances. If streams are implemented lazy, as is our intention, the instances will exist concurrently exploiting the producer/consumer parallelism within the task graph.

The function `powers` implements the task graph shown in Figure 5a. At one time it was thought that Sisal could not express such cyclic computations, but this is not correct. Sisal can express cyclic computations under three conditions: 1) the initial conditions of the input edges are known (i.e., the computation begins determinately), 2) the cyclic computation can be unrolled into a sequence of tasks, and 3) task output is synchronized with task input (i.e., a task can always wait for the next input before issuing the next output). The task graph in Figure 5a satisfies all three requirements.

The Sisal code for `powers` is given in the Appendix. Each iteration of the **`for initial`** expression removes the smaller of the values (call it `token`) at the head of **`old`** `s_stream` and **`old`** `b_stream`, appends the `token` to the output stream, defines a new `s_stream`, and defines a new `b_stream` with `token * prime` appended at the end. Since the task may exhaust `s_stream`, a test for `stream_empty` is necessary. On the other hand, the task can never exhaust `b_stream` since it appends a new value to the stream each iteration.

Notice how different the functional and imperative solution processes are. The functional programmer first specifies the computation logically and then translates the specification into code. He thinks in terms of expressions that, when executed, blossom into the needed task graphs. The scheduling of tasks, allocation of memory, and the synchronized access to shared data are implicit—the details are handled automatically by the compiler and runtime system. The imperative programmer might begin solving Hamming's Problem by first defining FIFO queues, a set of queue operations, and an access protocol for readers and writers. Then after writing code for the task bodies, he would explicitly wire together the tasks using the queues being careful not to violate the semantics of the operations or the access protocol. Insuring correctness is entirely his responsibility. The compiler and runtime system provide little or no support. Since it is easy to introduce subtle time-dependent errors into imperative parallel programs, the programming process is difficult, frustrating, and error prone.

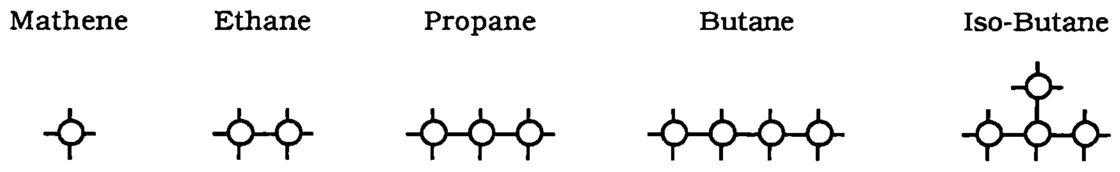


Figure 6 - Paraffins of size 1, 2, 3, and 4

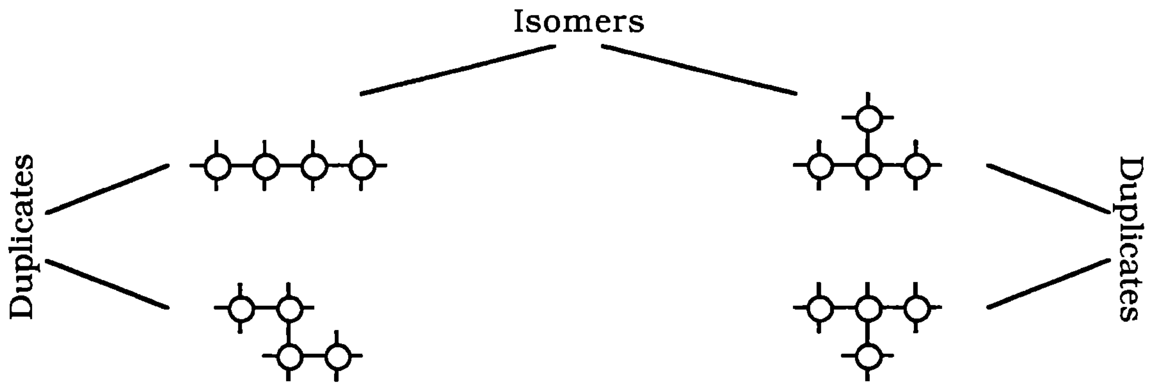


Figure 7 - Isomers and duplicates

4. The Paraffins Problem

4.1. A solution based on oriented trees

A paraffin is a hydrocarbon molecule with the chemical formula C_nH_{2n+2}. The Paraffins Problem asks us to output in increasing size the chemical structure of all paraffin molecules with n or fewer carbon atoms, including all isomers but no duplicates. Figure 6 show the paraffins of size 1, 2, 3, and 4. Since the placement of the carbon atoms uniquely defines the placement of the hydrogen atoms, we draw only the former. Isomers are different arrangements of the same number of carbon atoms. They have the same chemical formula, but different chemical properties. An isomer is a different arrangement of atoms, and not merely a rotation or reflection of a set of atoms (Figure 7).

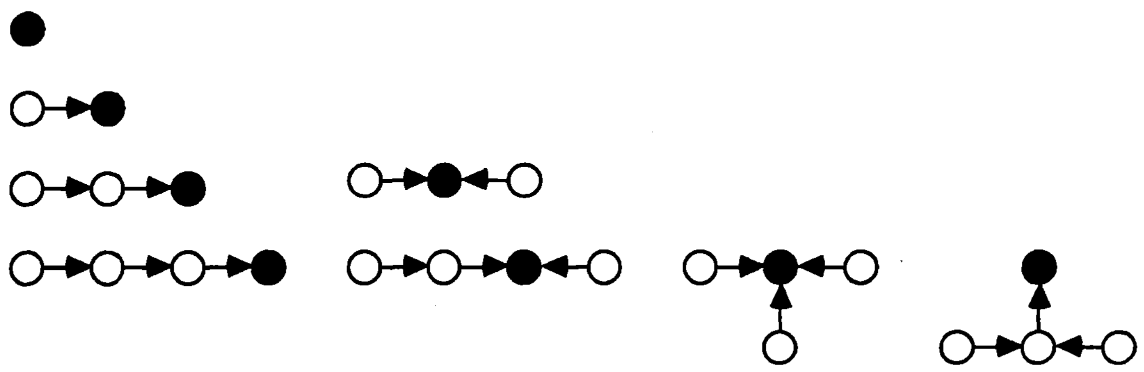


Figure 8 - Oriented trees of size 1, 2, 3, and 4

In [8], Turner presents a functional solution to the Paraffins Problem that first generates a list of paraffins and then filters out duplicates. Removing the duplicates is expensive and greatly increases the execution time of Turner's solution. A more efficient functional algorithm exists based on oriented and free trees [3]. Since this algorithm generates no duplicates, it does not require any post-processing.

An **oriented** tree is a connected, directed, acyclic graph. The tree includes a unique node called the **root**. All nodes other than the root are the source of a single arc, and there exists a unique path from every node to the root. The root is the source of no arc and is the sink of one or more arcs. The maximum number of arcs incident on any node is called the **fan-in**. Figure 8 shows the oriented trees of size 1, 2, 3, and 4 with fan-in less than 4. We define the relation, <, on oriented trees as: let T_{ij} be the j-th oriented tree of size i then

$$T_{ij} < T_{kl} \Rightarrow (i < k) \vee (i = k \wedge j < l).$$

Careful inspection of Figure 8 yields an efficient dynamic programming algorithm for constructing oriented trees of size n with fan-in three from the trees of size less than n with fan-in three. Logically, the algorithm is:

1. repeat for all choices of c, d, e, f, g, and h

2. draw a root with fan-in three (call the three edges left, bottom, and right)
3. choose three oriented trees T_{cd}, T_{ef}, and T_{gh}, possibly of size zero, such that

$$(c + e + g = n - 1) \wedge (T_{cd} \leq T_{ef} \leq T_{gh}) \qquad (4.1)$$

4. attach T_{cd}, T_{ef}, and T_{gh} to the left, bottom, and right edges, respectively.

Figure 9 illustrates how the algorithm constructs oriented trees of size four.

Paraffins with n or fewer carbons can be built from oriented trees of size less than or equal to $n/2$. The carbon atoms of a paraffin molecule form a **free tree**—an acyclic, connected graph with undirected edges. The tree's **centroid** is the node or nodes of minimum weight, where a node's **weight** is the size of its largest subtree. An important feature of a free tree is that its centroid is unique. If the number of nodes in the tree is odd, the centroid is a single node; if the number of nodes is even, the centroid is either a single node or two adjacent nodes. This fact motivates an efficient parallel algorithm for constructing paraffins of size n from the oriented trees of size less than $n/2$. Logically, the algorithm is:

Single Centroid:

```
1. repeat for all choices of $a$, $b$, $c$, $d$, $e$, $f$, $g$, and $h$
   2. draw a root with fan-in four (call the four edges top,
      left, bottom, and right)
   3. choose four oriented trees $T_{ab}$, $T_{cd}$, $T_{ef}$, and $T_{gh}$,
      possibly of size zero but less than n/2, such that
         $(a + c + e + g = n - 1) \wedge (T_{ab} \leq T_{cd} \leq T_{ef} \leq T_{gh})$     (4.2)
   4. attach $T_{ab}$, $T_{cd}$, $T_{ef}$,, and $T_{gh}$, to the top, left, bottom,
      and right edges, respectively.
```

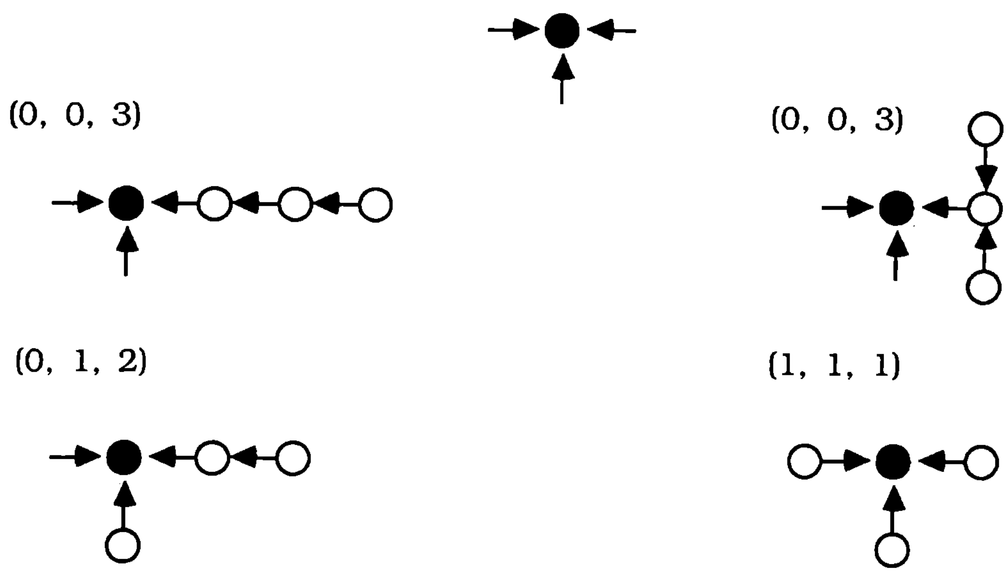


Figure 9 - Constructing oriented trees of size 4

Double Centroid:

1. repeat for all choices of b and d
   2. choose two oriented trees $T_{\frac{n}{2}b}$ and $T_{\frac{n}{2}d}$
   3. join together the roots of the two trees.

Figure 10 illustrates how the algorithm would construct paraffins of size 6.

The construction process must use oriented trees because it must consider each arrangement of nodes once per *distinguishable* set of nodes in the arrangement. For $n = 3$, there are two sets of distinguishable nodes: the end nodes and the interior node. There are two oriented trees: one whose root is an end node, and one whose root is the interior node. Note that there is only one free tree of size 3 (propane). Joining together the roots of the two oriented trees in all three ways—end node to end node, end node to interior node, and interior node to interior node—constructs the three isomers of size 6 with double centroid.

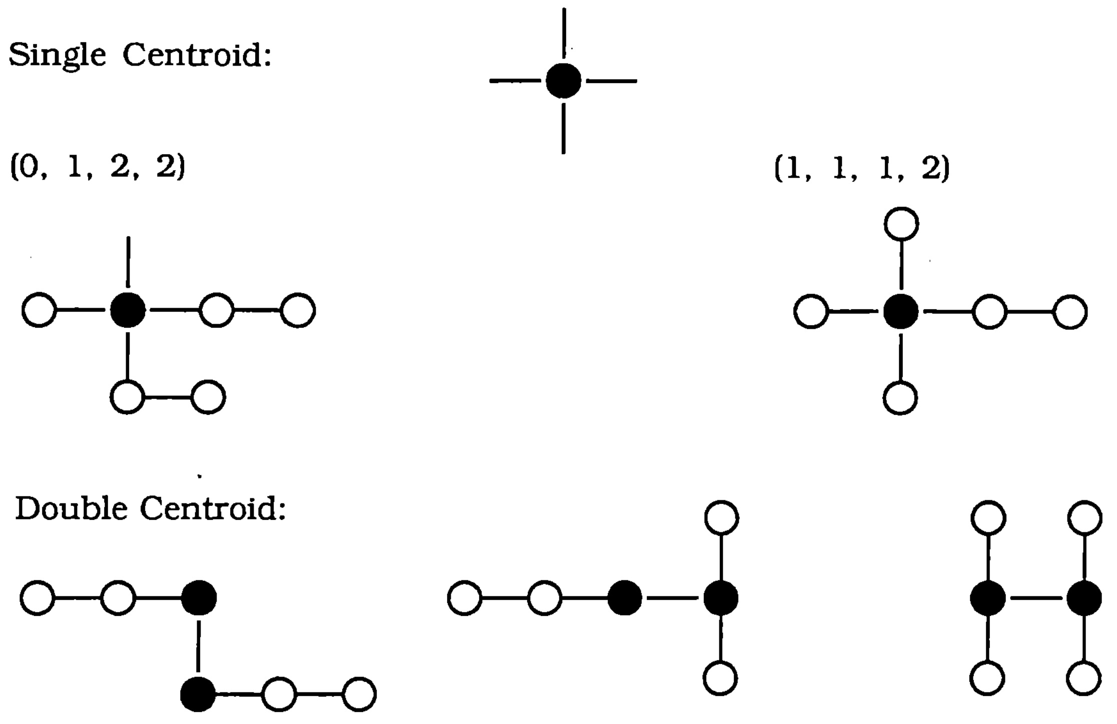


Figure 10 - Constructing paraffins of size 6

4.2. Constructing paraffins in Sisal

Writing Sisal functions to implement the algorithms described in the previous section is straightforward. We represent both oriented trees and free trees (paraffins) as character strings; i.e.,

```
    type trees = array[character]
```

For the first five paraffins (Figure 6), our program generates the character strings

```
    (C)    ((C)(C))    (C(C)(C))    ((C(C))(C(C)))    (C(C)(C)(C))
```

If the string is an oriented tree, then the first C is the root of the tree and the parenthesized lists of Cs at the same level as the root are the subtrees. If the string is a paraffin with a single centroid, the first C is the centroid. If the string is a paraffin with a double centroid, then the string divides into two strings with equal number of Cs. The centroid is the first C in each half.

We store the oriented trees and free trees (isomers) of size *i* as an array of trees,

```
type TreeArray1 = array[tree]
```

and the oriented trees and free trees (paraffin molecules) of size ≤ *n* as an array of arrays of type `tree`,

```
type TreeArray2 = array[TreeArray1]
```

The main function `Paraffins` takes an integer argument, `n`, and returns one value of type `TreeArray2`. Since the number of isomers is different for different number of carbon atoms, the result array will be ragged. The body of the function is the **let** expression,

```
let   Trees := OrientedTrees(n/2)
in    for i in 1, n
        isomer := if mod(i, 2) = 1 then
                    OneCentroid(i, Trees)
                  else
                    OneCentroid(i, Trees) ||
                    TwoCentroid(i, Trees)
                  end if
      returns array of  isomers
      end  for
end  let
```

The **let** clause defines the oriented trees and the **in** clause builds the paraffins of size 1 through *n*. The result array is built in parallel and returned in order.

The function `OrientedTrees` (see the Appendix) implements the dynamic programming algorithm described in the previous section. The function, a **for initial** expression, takes *n* as input and returns the oriented trees of size [0 ... $n/2$]. The initialization clause defines the oriented trees of size 0, 1, and 2, while the i-th iteration of the body, $3 \le i \le n/2$, builds the oriented trees of size i from the oriented trees of size less than i (i.e., **old** `Trees`). The trees are built by `XTrees` called from within the double-nested **for** expression

```
for c in 1, (i - 1)/3  cross
    e in c, (i - 1 - c)/2
```

We use the same names in the expression as in the logical description in the precious section. The ranges for `c` and `e` are set such that an instance of `XTrees` is invoked for each combination of tree sizes satisfying Equation 4.1. Since the value of `c` and `e` define the value of `g`, no third loop is required.

`XTrees` takes `a`, `c`, `e`, `g`, and `Trees` as input, and returns an array of trees. The function consists of three expressions (see Appendix). The first expression computes the cross product of the sets `Trees[c]` and `Trees[a]`, returning a array of array of tree pairs. The (b, d)-th element of the array is the pair {`Trees[c,d]`, `Trees[a,b]`}. The second expression computes the cross product of the sets `Trees[e]` and `Trees[g]`. The third expression then takes the results of the first two expressions, computes their cross product, and builds the new trees.

An important feature of the function is that it creates no duplicates. The caller guarantees that `a` ≤ `c` ≤ `e` ≤ `g`; therefore, duplicates can occur only if two adjacent parameters are equal. In which case, the cross product of their sets is symmetric. We can eliminate the symmetry, and thus the duplicates, by structuring the cross products as

(Trees[c] x Trees[a]) x (Trees[e] x Trees[g])

and including the following tests:

1. if `c` = `a`, then return only the lower triangle of the cross product (Figure 11a); else, return the entire product.

Trees[a] = {x, y, z}   Trees[e] = {x, y, z}
Trees[c] = {x, y, z}   Trees[f] = {x, y, z}

```
(x,x)
(y,x)  (y,y)
(z,x)  (z,y)  (z,z)
```

(a)

```
(x,x)  (x,y)  (x,z)
       (y,y)  (y,z)
              (z,z)
```

(b)

```
(x,x,x,x)  (x,x,x,y)  (x,x,x,z)

(y,x,x,x)  (y,x,x,y)  (y,x,x,z)  (y,x,y,y)  (y,x,y,z)
(y,y,x,x)  (y,y,x,y)  (y,y,x,z)  (y,y,y,y)  (y,y,y,z)

(z,x,x,x)  (z,x,x,y)  (z,x,x,z)  (z,x,y,y)  (z,x,y,x)  (z,x,z,z)
(z,y,x,x)  (z,y,x,y)  (z,y,x,z)  (z,y,y,y)  (z,y,y,z)  (z,y,z,z)
(z,z,x,x)  (z,z,x,y)  (z,z,x,z)  (z,z,y,y)  (z,z,y,z)  (z,z,z,z)
```

(c)

Figure 11 - The results of the expressions in Xtrees for a = c = e = g

2. if e = g, then return only the upper triangle of the cross product (Figure 11b); else, return the entire product.
3. if c = e, then return only the lower triangle of the cross product (Figure 11c); else, return the entire product.

The function `OneCentroid` (see the Appendix) takes `i` and the array of oriented trees, and returns the free trees of size `i` with one centroid. The triple-nested range generator

```
for a in 0, (i - 1)/4     cross
    c in a, (i - 1 - a)/3 cross
    e in max(c, i/2 - a - c), (i - 1 - a - c)/2
```

creates an instance of the body (a call to `XTrees`) for all combinations of tree sizes satisfying Equation 4.2. Since the values of a, c, and e set the value of `g`, no fourth loop is required. `TwoCentroid` (see the Appendix) takes `i` and the array of oriented trees, and returns the free trees of size `i` with a double centroid. The double-nested range generator

```
    for b in 1, array_size(Trees[i/2]) cross
        d in b, array_size(Trees[i/2])
```

spawns an instance of the array build operation for every pair of oriented trees of size $i/2$. Both functions build their respective tree structures without duplicates and in order.

Our solution relies heavily on the ragged structure of Sisal's arrays and the semantics of the **for** expression's range generator and returns clause. With careful thought and organization, we have been able to implement the complicated combinatorical requirements of the problem without generating duplicates. The Sisal solution is parallel, and executes in minimal space and time.

5. The Doctor's Office

5.1. A logical view of the Doctor's Office

Given a list of patients and doctors model the following system. Originally, all patients are well and all doctors are available. Doctors await patients at their office in a FIFO queue. At random times, patients become sick, travel to the doctor's office, and wait in a FIFO queue to see a doctor. A nurse pairs the first patient and the first doctor in line, and assigns them an examination room. In the examination room, the doctor cures the patient in a random amount of time. The patient then rejoins the world and the doctor returns to the nurse's station. Figure 12 shows a logical view of the Doctor's Office comprised of three tasks and four edges. The edges act as FIFO queues of zero or more values.

As in Hamming's Problem, we are faced with implementing a cyclic computation. Again we use a **`for initial`** expression and rely on the expression's loop semantics to satisfy the cyclic dependencies. We use arrays, not streams, to implement the FIFO queues for reasons that will become apparent shortly. Unlike Hamming's Problem, only one of the three conditions listed in Section 3.2 is satisfied—the initial condition of each edge is known.

Patients_In	- empty
Patient_Out	- the list of patients
Doctor_Out	- the list of doctors
Patient_Doctor	- empty

The semantics of the **`for initial`** expression implies a sequence of tasks and synchronization of task input/output that violates the spirit, if not the letter, of the specification. Ideally, the three tasks should execute and communicate asynchronously without any constraints.

Unfortunately, Sisal excludes all forms of asynchrony. Its functions are determinate. Outputs depend **only** on inputs regardless of architecture, operating system, system load, or program state. The consumer of a stream cannot test the stream for data availability; if so, the consumer could be programmed to take different actions depending on whether or not data had arrived. The function's outputs would then depend on the execution speed of the producer, the speed and congestion of the communication network, and the scheduling policy of the operation system. Once the consumer tries to read the next stream value, it must wait until that value arrives. To solve the Doctor's Office to the letter of the specification, the three tasks must continue to execute whether or not new data arrives on all their input edges. That is, patients must become sick whether or not cured patients return from the doctor's office, patient–doctors pairs must leave the nurse's station whether or not new patients or doctors arrive, and patients must be cured whether or not new patient–doctors pairs be-

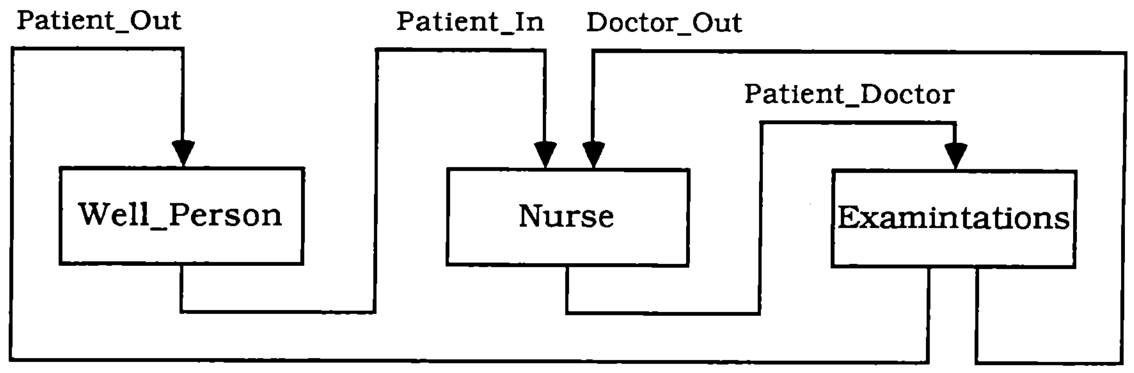


Figure 12 - A logical view of the Doctor's Office Problem

gin treatment. In Section 5.3 we address this dilemma and propose a reasonable solution.

5.2. The main function

We define two new types

```
type queue  = array[integer];
type queue2 = array[array[integer]];
```

and write the main function as

```
for initial
  seed                     := 0;
  patient_in               := array queue [];
  patient_out              := list_of_patients;
  doctor_out               := list_of_doctors;
  patient_doctor           := array queue2 []
while true repeat
  seed                     := next_seed(old seed);
  patient_in               := well_person(seed, old patient_out);
  patient_doctor           := nurse(old patient_in, old doctor_out);
  patient_out, doctor_out  := examinations(seed, old patient_doctor)
```

```
returns stream of patient_in
        stream of doctor_out
end for
end function
```

patient_in, patient_out, and doctor_out are of type queue; and doctor_patient is of type queue2. The initialization segment defines the initial values of the four arrays and seed. The latter is used by well_person and examinations to drive a random number generator. The three tasks in the body execute independently consuming the edge values defined on the previous iteration. The expression returns a stream of the patient queues, and a stream of the doctor queues.

The expression as written is not correct. Notice that well_person takes as input **old** patient_out, the patient(s) cured on the previous iteration and who are now rejoining the array of well patients. All the patients that were well on the previous iteration and did not fall sick are lost. The state has not been retained. The well patients, the patients and doctors waiting at the nurse's station, and the patients and doctors in the examination rooms are persistent sets of data which are not created and consumed in a single action. Since Sisal functions are side-effect free and do not retain state between invocation, we must explicitly maintain and circulate the state from iteration to iteration.

Thus well_person must return two sets or arrays: the new sick patients, and the array of patients that are still well. nurse must return three arrays: the new patient-doctor pairs, sick patients still waiting for doctors, and available doctors still waiting for patients. examinations must return three arrays: newly cured patients, newly available doctors, and patient-doctor pairs still in the examination rooms. Figure 13 shows the new edges (arrays) and the catenate operations necessary to reassemble state. The new main function is

```
for initial
  seed              := 0;
  still_well        := list_of_patients;
  patient_out       := array queue [];
  still_sick        := array queue [];
  patient_in        := array queue [];
  still_available   := list_of_doctors;
  doctor_out        := array queue [];
  still_examining   := array queue2 [];
  patient_doctor    := array queue2 []
while true repeat
  seed              := next_seed(old seed);

  still_well, patient_in :=
     well_person(seed, old still_well || old patient_out);

  still_sick, patient_doctor, still_available :=
     nurse(old still_sick      || old patient_in,
           old still_available || old doctor_out);

  still_examining, patient_out, doctor_out :=
     examinations(seed, old still_examining || old patient_doctor)

returns stream of patient_in
        stream of doctor_out
end for
```

For simplicity, we have pushed the catenate operations into the parameter lists of the functions.

5.3. The three tasks

Tasks of iteration i cannot execute until the tasks of iteration $(i-1)$ have completed. The loop-carried dependencies impose a constraint on task execution not specified in the problem description. As explained in Section 5.1, the tasks of iteration $(i-1)$ must generate some output for the tasks of iteration i to execute; otherwise, those

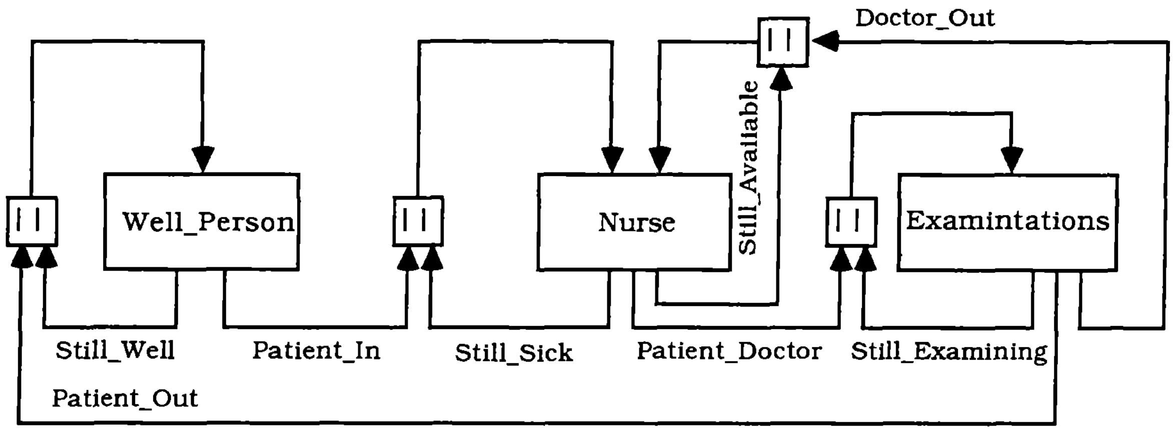


Figure 13 - The Sisal solution to the Doctor's Office Problem

tasks will hang waiting for data. If we insist that a patient becomes sick or is cured every iteration, we would further violate the problem's specifications. Instead, we force each task to issue either an empty array (a *ghost*) of the appropriate type, or a single-element array. Issuing single-element arrays is not a constraint imposed by the language. We could have written the tasks to issue any number, even a random number, of patients or patient-doctor pairs. Empty arrays are removed automatically from the system by the catenate operations.

The code for `well_person` is

```
function well_person (seed: integer; patients: queue
              returns  queue, queue)
  let
    x    := random(seed);
    size := array_size(patients);
    sick := floor(real(size) * x / 0.7) + 1
  in
    if size = 0 | x >= 0.7 then
      patients, array queue []
```

```
        else
          array_remh(patients[sick: patients[size]]),
          array [1: patients[sick]]
        end if
      end let
    end function
```

The `let` clause defines a random number `x`, computes the number of well patients, and the index of the patient who may fall sick (call him Bob). If there are no well patients (`size = 0`) or no patient falls sick (`x >= 0.7`), the function returns the input array `patients` and an empty array. Otherwise, the function removes Bob from the array of well patients by "replacing" his identification number with the identification number of the last person in the array, and then removing the last person. The resulting array is returned as the function's first result. Bob's identification number is placed in an array and returned as the function's second result.

The code for `nurse` and `examinations` is similar. We refer the reader to the Appendix.

While our solution to the Doctor's Office is not perfect, it is close. Since Sisal explicitly excludes all forms of asynchrony, we could never hope to solve the problem exactly. The fact that we came as close as we did to modeling a real doctor's office is a testimony to Sisal's robustness and generality.

6.0. Skyline Matrix Problem

6.1. Gaussian elimination without pivoting

In this problem, we are asked to solve the linear system of equations

$$A\,x = b \qquad (6.1)$$

without pivoting where A is a **skyline matrix**. A skyline matrix has nonzero values in row i in columns k through i, $1 \leq k \leq i$, and nonzero values in column j in rows k through j, $1 \leq k \leq j$. The values of k are stored in two vectors: *row* and *column*. Figure 14 depicts a skyline matrix and its associated *row* and *column* vectors. Notice how the nonzero values form a skyline both above and below the diagonal.

A traditional solution method for linear systems of equations is Gaussian elimination without pivoting. The method reduces A to the triangular matrix A', and b to the column vector b' such that

$$A' x = b' \tag{6.2}$$

A' can be either upper or lower triangular. Equation 6.2 is then solved using a method known as *back substitution*.

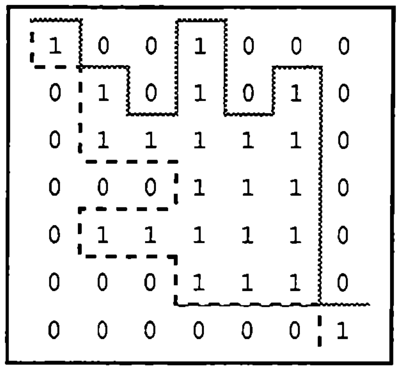
The Sisal algorithm developed in Section 6.3 forms a lower triangular matrix. The reduction occurs in $n - 1$ steps, one step per row. To eliminate the upper triangular matrix we step backwards from n to 1. At step i, we reduce the system as follows

$$A_{j,k} = A_{j,k} - (A_{i,k} \bullet A_{j,i} \ / \ A_{i,i}), \qquad 1 \leq j < i, \ 1 \leq k \leq i \tag{6.3.1}$$

and

$$b_j = b_j - (b_i \bullet A_{j,i} \ / \ A_{i,i}), \qquad 1 \leq j < i \tag{6.3.2}$$

The elements in rows i through n, and in columns $i + 1$ through n remain unchanged. The reduction's effect is to set the values in column i above the diagonal to zero, thereby, eliminating the i-th column in the upper triangular matrix. Row i and column i are referred to as the **pivot row** and **pivot column**, respectively. $A_{i,i}$ and b_i are referred to as the **pivot element** of A and the **pivot element** of b, respectively. On completion, A' will consist of the n pivot rows, and b' will consist of the n pivot elements of b.



A

row = [1, 2, 2, 4, 2, 4, 7] column = [1, 2, 3, 1, 3, 2, 7]

Figure 14 - A skyline matrix and associated row and column vectors

To solve for x, we first solve for x_1

$$x_1 = b'_1 \;/\; A'_{1,1} \tag{6.4.1}$$

Once we have x_1, we can solve for x_2

$$x_2 = (b'_2 - A'_{2,1}\, x_1) \;/\; A'_{2,2} \tag{6.4.2}$$

and then for x_3

$$x_3 = (b'_3 - A'_{3,1}\, x_1 - A'_{3,2}\, x_2) \;/\; A'_{3,3} \tag{6.4.3}$$

and so on.

6.2. An efficient representation of a skyline matrix in Sisal

The key to an efficient implementation of the Skyline Matrix Problem is eliminating the zeros at the beginning of each row and column, and eliminating the computations involving those zeros. Sisal's ragged array structure is ideally suited for this problem. Since a "two-dimensional" array in Sisal is an array of arrays, and since each component array can have a different size and lower bound, we can eliminate the zeros at the head of each row by setting the lower bound of component *i* to `row[i]`. Recall that `row` and `column` store the location of the first non-zero element in each row and column. Since the component arrays must be continuous, we can eliminate some, but not all, of the zeros above the diagonals by setting the upper bound of each component array to the index location of the last non-zero element in the row. For example, the zeros in columns 5 through 7 of the first row in the array shown in Figure 14 can be eliminated by setting the upper bound of row 1 to 4; however the zeros in column 2 and 3 remain.

We can eliminate all the zeros by splitting *A* into its lower and upper triangular submatrices and transposing the upper. We refer to the former as `L` and the latter as `U` (even though it is really U-transpose). Under this decomposition, all leading zeros fall at the head of rows and can be eliminated by setting the lower bound of the *i*-th component of `L` and `U` to `row[i]` and `column[i]`, respectively. Figure 15 shows such a decomposition. The Sisal code to build the two arrays is

```
L := for i in 1, n cross j in row[i], i
     returns array of A[i, j]
     end for;
U := for i in 1, n cross j in column[i], i
      returns array of A[j, i]
      end for;
```

Both expressions return a "two-dimensional" array of `n` rows. The lower bound of each row is the first value of the inner range, either `row[i]` or `column[i]`, and the upper bound is `i`; thus, each row stores only nonzero elements. The transpose in forming `U` is effected by reversing `i` and `j` in the read.

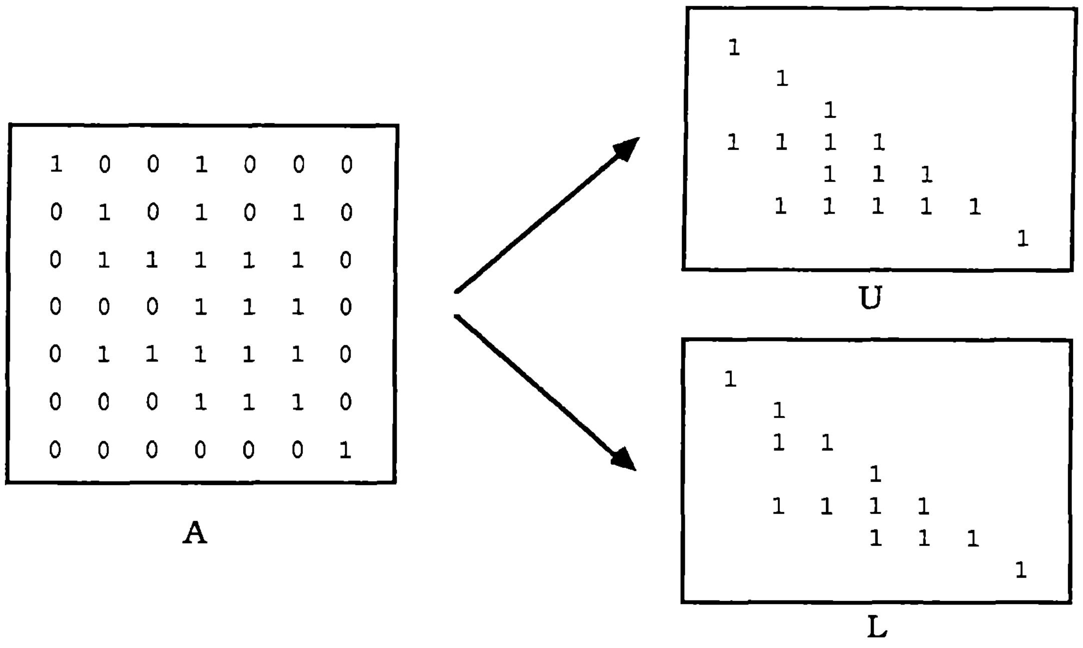


Figure 15 - A skyline matrix decomposed into L and U

6.3. An efficient algorithm based on the Sisal decomposition

Having eliminated all the zeros that lie outside the skyline, we must now develop an efficient algorithm that takes `L`, `U`, and `b`, and returns `A_prime` and `b_prime`. Eliminating the lower triangle of *A* would require us to work with the columns of `L` and `U`. Since Sisal arrays are row-oriented, column-oriented algorithms are more complex and less efficient. Instead, if we eliminate the upper triangle of *A*, we will work with the rows of `L` and `U`, a much easier and more straightforward proposition.

The following **`for initial`** expression implements the iterative algorithm described in Section 6.1,

```
A_prime, b_prime :=
   for initial
     i           := n;
     pivot_b     := b[n];
     pivot_A     := L[n];
     L1, U1, b1  := reduce(n, L, U, b)
   while i > 1 repeat
     i           := old i - 1;
     pivot_b     := old b1[i];
     pivot_A     := old L1[i];
     L1, U1, b1  := reduce(i, old L, old U, old b)
   returns array of pivot_A
           array of pivot_b
   end for
```

The expression steps backwards in single steps from *n* to 1. Each iteration defines a new value of `L1`, `U1`, and `b1` "reduce"d from the arrays' previous values, and contributes one element to `A_prime` and `b_prime`. The contributed values are: `L1[i]`, the *i*-th pivot row, and `b1[i]`, the *i*-th pivot element of *b*.

A common mistake that novice Sisal programmers make is to build `A_prime` and `b_prime` in the body one element at a time, and carry the partial arrays from iteration to iteration. They append each iteration's contribution to partial built arrays using `array_addh`,

```
A_prime := array_addh(old A_prime, A_pivot);
b_prime := array_addh(old b_prime, b_pivot);
```

This complicates the expression, and is not necessary. Novices fail to understand that Sisal expressions return values, including arrays, as a consequence of their execution. They continue to think imperatively, describing both the "what" and "how" of the computation.

The function `reduce` implements Equations 6.3.1 and 6.3.2. Rewriting the expressions in terms of *L* and *U*, we have

$$L_{j,k} = L_{j,k} - (L_{i,k} \bullet U_{i,j} \;/\; L_{i,i}) \qquad (6.5.1)$$
$$U_{j,k} = U_{j,k} - (L_{i,j} \bullet U_{i,k} \;/\; L_{i,i}) \qquad (6.5.2)$$
$$b_j = b_j - (b_i \bullet U_{i,j} \;/\; L_{i,i}) \qquad (6.5.3)$$

where $1 \le j < i$ and $1 \le k \le j$. We can express the three equations in Sisal as

```
L1 := for j in 1, i - 1 cross k in 1, j
        returns array of
          old L1[j,k] - (old L1[i,k] * old U1[i,j] / old L1[i,i])
        end for;
U1 := for j in 1, i - 1 cross k in 1, j
        returns array of
          old U1[j,k] - (old L1[i,j] * old U1[i,k] / old L1[i,i])
        end for;
b1 := for j in 1, i - 1
        returns array of
          old b1[j] - (old b1[i] * old U1[i,j] / old L1[i,i])
        end for;
```

Notice that `L1`, `U1`, and `b1` are one element smaller than their **old** counterparts. Because **old** `L1` and **old** `U1` are compressed some elements may be missing. Reading a missing element will return an error value, which we can test for error using the intrinsic function `is error`. `is error(x)` returns **true** if `x` is an error value; else, it returns **false**.

If **old** `U1[i,j]` is missing (i.e., the pivot column element is zero), Equations 6.5.1 and 6.5.3 reduce to

$$L_{j,k} = L_{j,k}$$
$$b_j = b_j$$

for $1 \le k \le j$. If the value is present, then elements of the j-th row of L will change, but only elements from the minimum of the lower bounds of **old** `L1[i]` and **old** `L1[j]` to `j` will change. The other elements are zero and will remain zero. If **old** `L1[i,j]` is missing (i.e., the pivot row element is zero), Equation 6.5.2 reduces to

$$U_{j,k} = U_{j,k}$$

for $1 \le k \le j$. If the value is present, then elements of the j-th row of U will change, but only elements from the minimum of the lower bounds of **old** `U1[i]` and **old** `U1[j]` to `j` will change. The other elements are zero and will remain zero. The Appendix gives the revised Sisal expression for `L1`, `U1`, and `b1` in **function** `reduce`.

Once we have `A_prime` and `b_prime`, we can solve for x according to Equations 6.4. The Sisal code is straightforward and we leave it as an exercise for the reader (see the Appendix for our solution).

Despite Sisal's high-level functional semantics, we have developed an efficient solution of the Skyline Matrix Problem which stores only nonzero elements and avoids all computation involving zero elements. Moreover, at any time, the program's data structures store only essential information. The belief that functional languages are unable to express scientific computations, and that array operations in these languages are unnatural and inefficient is just not true. The Sisal code is more efficient, easier to understand, and certainly closer to the mathematics of the problem than the Fortran solution presented in [2].

7.0 Conclusions

In this chapter we have presented Sisal solutions to the four Salishan problems. All the solutions are parallel, and preliminary studies show that compile-time analysis [1, 6] can eliminate all unnecessary copying and memory management operations. We expect these solutions to execute as fast as imperative solutions on conventional multiprocessor systems. We were able to meet problem specifications in three of the four cases. In the odd case, the Doctor's Office, the nondeterministic nature of the problem prevents us from implementing the problem exactly as specified. Given the havoc which unintended nondeterminism wreaks in the development of correct parallel programs, we do not apologize for the "shortcoming" of determinacy in Sisal.

We wish to leave the reader with three important facts. First, Sisal can express a wide variety of problems, not just scientific computations. The language supports a robust set of array operations without violating functional semantics.

Second, functional programming is more abstract than imperative programming. Since Sisal programs encode only the "what" and not the "how" of problem solutions, the Sisal programmer is not encumbered by many tedious and picayune details. The mathematical semantics of Sisal provide the scientific programmer with a natural and familiar medium in which to express his computations. We do not want to teach scientists yet another language; on the contrary, we want to return them to their mathematical roots.

Third and most important, all four Sisal programs are parallel, determinate, and deadlock free. The codes will run on any computer system, regardless of topology or number of processors, in parallel and without rewriting. Yet at no time did we ever think about parallelism, communication, synchronization, or task scheduling. We simply designed a mathematical solution to the problem, and then translated it into Sisal code. Compare this to the imperative solutions presented in this book in which considerable time and effort is expended managing parallelism, communication, synchronization, and task scheduling. Truly, writing parallel programs in a functional language is free.

Acknowledgements

I would like to thank Steven Skedzielewski who worked with me on designing the Sisal solutions for presentation at the 1988 Salishan

Conference, and Tom DeBoni who proofread the chapter and suggested numerous ways to improve the presentation of the algorithms.

This work was supported (in part) by the Applied Mathematics Program of the Office of Energy Research (U.S. Department of Energy) under contract No. W-7405-Eng-48 to Lawrence Livermore National Laboratory.

References

1. Cann, D. C. *Compilation Techniques for High Performance Applicative Computation.* Ph.D. thesis, Department of Computer Science, Colorado State University, 1989.
2. Eisenstat, S. C. and A. H. Sherman. *Subroutines for envelope solution of sparse linear systems.* Research Report 35, Yale University, New Haven, CT, October 1974.
3. Knuth, D. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1973.
4. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.1.* Lawrence Livermore National Laboratory Manual M-146, Lawrence Livermore National Laboratory, Livermore, CA, June 1983.
5. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2.* Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.
6. Ranelletti, J. E. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* Ph.D. thesis, Department of Computer Science, University of California at Davis/Livermore, 1987.
7. Skedzielewski, S. K. and J. Glauert. *IF1 - An intermediate form for applicative languages.* Lawrence Livermore National Laboratory Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.
8. Turner, D. A. The semantic elegance of applicative languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, NH, October 1981, 85–92.

Appendix

```
/************************************/
/**   Hamming's Problem, Extended   **/
/************************************/

define hamming

type OneDim  = array[integer];
type Istream = stream[integer];

function powers(n, prime: integer; in_stream: Istream
        returns Istream)

  for initial
      token    := stream_first(in_stream);
      s_stream := stream_rest(in_stream);
      b_stream := stream [token * prime]
  while token < n repeat
      token, s_stream, b_stream :=
        let
          s_token := stream_first(old s_stream);
          b_token := stream_first(old b_stream)
        in
          if stream_empty(old s_stream) then
             b_token,
             old s_stream,
             stream_append(stream_rest(old b_stream), b_token * prime)
          elseif b_token < s_token then
             b_token,
             old s_stream,
             stream_append(stream_rest(old b_stream), b_token * prime)
          else
             s_token,
             stream_rest(old s_stream),
             stream_append(old b_stream, s_token * prime)
          end if
        end let
  returns stream of token when token <= n
  end for

end function % powers

function hamming (n: integer; primes: OneDim
          returns Istream)

   for initial
       i        := 0;
       s_stream := stream [1]
   while i < array_size(primes) repeat
```

```
        i        := old i + 1;
        s_stream := powers(n, primes[i], old s_stream)
    returns value of s_stream
    end for

end function % hamming



/**************************/
/**   Paraffins Problem   **/
/**************************/

define Paraffins

type trees      = array [character];
type TreeArray1 = array [trees];
type TreeArray2 = array [TreeArray1];

function Xtrees(a, c, e, g: integer; Trees: TreeArray2
        returns TreeArray1)

  let
      CxA := if c = a then
                 for d in 1, array_size(Trees[c]) cross
                     b in 1, d
                 returns array of Trees[c, d] || Trees[a, b]
                 end for
             else
                 for d in 1, array_size(Trees[c]) cross
                     b in 1, array_size(Trees[a])
                 returns array of Trees[c, d] || Trees[a, b]
                 end for
             end if;

      ExG := if e = g then
                 for f in 1, array_size(Trees[e]) cross
                     h in f, array_size(Trees[e])
                 returns array of Trees[e, f] || Trees[g, h]
                 end for
             else
                 for f in 1, array_size(Trees[e]) cross
                     h in 1, array_size(Trees[g])
                 returns array of Trees[e, f] || Trees[g, h]
                 end for
             end if
  in
     if c = e then
        for d in 1, array_size(CxA)    cross
            b in 1, array_size(CxA[d]) cross
            f in 1, d                  cross
            h in 1, array_size(ExG[f])
         returns value of catenate
            array [1: "(C" || CxA[d, b] || ExG[f, h] || ")"]
         end for
      else
         for d in 1, array_size(CxA)    cross
            b in 1, array_size(CxA[d]) cross
            f in 1, array_size(ExG)    cross
            h in 1, array_size(ExG[f])
         returns value of catenate
            array [1: "(C" || CxA[d, b] || ExG[f, h] || ")"]
         end for
      end if
   end let

end function % Xtrees

function OrientedTrees(n: integer returns TreeArray2)

   for initial
      i     := 2;
      Trees := array [0: array [1: ""],
                         array [1: "(C)"],
                         array [1: "(C(C))"]]
   while i < n repeat
      i     := old i + 1;
      set_i := for c in 0, (i - 1) / 3 cross
                   e in c, (i - 1 - c) / 2
                   g := i - 1 - c - e
               returns value of catenate Xtrees(0, c, e, g, old Trees)
               end for;
      Trees := array_addh(old Trees, set_i)
   returns value of Trees
   end for

end function % Oriented Trees

function OneCentroid(i: integer; Trees: TreeArray2
             returns TreeArray1)

   for a in 0, (i - 1) / 4     cross
      c in a, (i - 1 - a) / 3 cross
      e in max(c, i/2 - a - c), (i - 1 - a - c) / 2
      g := i - 1 - a - c - e
   returns value of catenate Xtrees(a, c, e, g, Trees)
   end for

end function % OneCentroid

function TwoCentroid(i: integer; Trees: TreeArray2
             returns TreeArray1)
```

```
  for b in 1, array_size(Trees[i/2]) cross
      d in b, array_size(Trees[i/2])
  returns value of catenate
      array [1: "(" || Trees[i/2, b] || Trees[i/2, d] || ")"]
  end for

end function % TwoCentroid

function Paraffins(n: integer returns TreeArray2)

  let
      Trees := OrientedTrees(n / 2)
  in
       for i in 1, n
         isomer := if mod(i, 2) = 1 then
                      OneCentroid(i, Trees)
                   else
                      OneCentroid(i, Trees) || TwoCentroid(i, Trees)
                   end if
       returns array of isomer
       end for
   end let

end function % Paraffins



/*********************************/
/**   Doctor's Office Problem   **/
/*********************************/

define doctors_office

type queue  = array [integer];
type queue2 = array [array [integer]];

global random    (seed: integer  returns real)
global next_seed (seed: integer  returns integer)

function well_persons (seed: integer; patients: queue
               returns queue, queue)

  let
      x    := random(seed);
      size := array_size(patients);
      sick := floor(real(size) * x / 0.7) + 1
  in
      if size = 0 | x >= 0.7 then
         patients, array queue []
      else
```

```
         array_remh(patients[sick: patients[size]]),
         array [1: patients[sick]]
      end if
   end let

end function % well_persons

function nurse(patients: queue; doctors: queue
       returns queue, queue2, queue)

   let
      n_patients := array_size(patients);
      n_doctors  := array_size(doctors)
   in
      if (n_patients = 0) | (n_doctors = 0) then
         patients, array queue2 [], doctors
      else
         array_setl(array_reml(patients), 1),
         array [1: array [1: patients[1], doctors[1]]],
         array_setl(array_reml(doctors), 1)
      end if
   end let

end function % nurse

function examinations (seed: integer; in_exam: queue2
              returns queue2, queue, queue)

   let
      x     := random(seed);
      size  := array_limh(in_exam);
      cured := floor(real(size) * x / 0.3) + 1
   in
      if size = 0 | x >= 0.3 then
         in_exam, array queue [], array queue []
      else
         array_remh(in_exam[cured: in_exam[size]]),
         array [1: in_exam[cured, 1]],
         array [1: in_exam[cured, 2]]
      end if
   end let

end function % examinations

function doctors_office (list_of_patients, list_of_doctors: queue
                 returns stream[queue], stream[queue])

   for initial
     seed            := 0;
     still_well      := list_of_patients;
     patient_out     := array queue [];
```

```
    still_sick       := array queue [];
    patient_in       := array queue [];
    still_available := list_of_doctors;
    doctor_out       := array queue [];
    still_examining := array queue2 [];
    patient_doctor  := array queue2 []
  while true repeat
    seed             := next_seed(old seed);
    still_well, patient_in :=
       well_persons(seed, old still_well || old patient_out);
    still_sick, patient_doctor, still_available :=
       nurse(old still_sick      || old patient_in,
             old still_available || old doctor_out);
    still_examining, patient_out, doctor_out :=
       examinations(seed, old still_examining || old patient_doctor)
  returns stream of patient_in
          stream of doctor_out
  end for

end function % doctors_office



/*********************************/
/**   Skyline Matrix Problem   **/
/*********************************/

define skyline

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

function form_skyline(n: integer; row, column: IntDim; A: TwoDim
              returns TwoDim, TwoDim)

  for i in 1, n cross j in row[i], i
  returns array of A[i, j]
  end for,

  for i in 1, n cross j in column[i], i
  returns array of A[j, i]
  end for

end function % form_skyline

function reduce(i: integer; L1, U1: TwoDim; b1: OneDim
        returns TwoDim, TwoDim, OneDim)

  % reduce L1
  for j in 1, i - 1 returns array of
```

```
    if is error(U1[i, j]) then
       L1[j]
    else
       for k in min(array_liml(L1[i]), array_liml(L1[j])), j
       returns array of
         if is error(L1[i, k]) then
            L1[j, k]
         elseif is error(L1[j, k]) then
            - (L1[i, k] * U1[i, j]) / L1[i, i]
         else
            L1[j, k] - (L1[i, k] * U1[i, j]) / L1[i, i]
         end if
       end for
    end if
  end for,

  % reduce U1
  for j in 1, i - 1 returns array of
    if is error(L1[i, j]) then
       U1[j]
    else
       for k in min(array_liml(U1[i]), array_liml(U1[j])), j
       returns array of
         if is error(U1[i, k]) then
            U1[j, k]
         elseif is error(U1[j, k]) then
            - (L1[i, j] * U1[i, k]) / L1[i, i]
         else
            U1[j, k] - (L1[i, j] * U1[i, k]) / L1[i, i]
         end if
       end for
    end if
  end for,

  % reduce b1
  for j in 1, i - 1 returns array of
    if is error(U1[i, j]) then
       b1[j]
    else
       b1[j] - (b1[i] * U1[i, j]) / L1[i, i]
    end if
  end for

end function % reduce

function eliminate(n: integer; L, U: TwoDim; b: OneDim
           returns TwoDim, OneDim)

  for initial
      i          := n;
      pivot_b    := b[n];
```

```
        pivot_A     := L[n];
        L1, U1, b1 := reduce(n, L, U, b)
    while i > 1 repeat
        i           := old i - 1;
        pivot_b     := old b1[i];
        pivot_A     := old L1[i];
        L1, U1, b1 := reduce(i, old L1, old U1, old b1)
    returns array of pivot_A
            array of pivot_b
    end for

  end function % eliminate

  function backsolve(n: integer; A_prime: TwoDim; b_prime: OneDim
             returns OneDim)

    for initial
        i := n;
        j := 1;
        A := A_prime;
        b := b_prime;
        x := b[n] / A[n, 1]
    while i > 1 repeat
        i := old i - 1;
        j := old j + 1;
        b := for k in 1, i
               b_k := if is error(A_prime[k, old j]) then
                          old b[k]
                      else
                          old b[k] - old x * A_prime[k, old j]
                      end if
             returns array of b_k
             end for;
        x := b[i] / A[i, j]
    returns array of x
    end for

  end function % backsolve

  function skyline(n: integer; row, column: IntDim;
                   A: TwoDim;  b: OneDim
           returns OneDim)

    let
        L, U := form_skyline(n, row, column, A);
        A_prime, b_prime := eliminate(n, L, U, b)
    in
        backsolve(n, A_prime, b_prime)
    end let

  end function % skyline
```