M-170

IF1
An Intermediate Form for Applicative Languages

July 31, 1985

VERSION 1.0

Table of Contents

Figures

Tables

IF1
An Intermediate Form for Applicative Languages

Stephen Skedzielewski[1]
John Glauert[2]

July 31, 1985

The intent of IF1 is to define a language that can be the target of several compilers for applicative languages. We hope to be able to exchange programs translated from different sources via IF1, and share the rather limited resource of programs written in applicative languages. The language that we define is not sufficiently general to be used by implementors of all applicative languages; it is strongly based on the features of SISAL [McGraw 1985] and VAL [Ackermann 1979]. It is our hope that IF1 will be sufficient to form the basis for other, more comprehensive intermediate forms for a larger set of applicative languages.

[1]Computing Research Group, Lawrence Livermore National Laboratory, P. O. Box 808, L-306, Livermore, CA 94550

[2]School of Information Science, University of East Anglia, Norwich NR4 7TJ, United Kingdom

This work was supported (in part) 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.

1. Background and Acknowledgements

IF1 began as an attempt to define an intermediate language for a higher-level data flow language (SISAL) in late 1982. As interest developed at Lawrence Livermore National Laboratory, the University of Manchester, Digital Equipment Corporation and Colorado State University to develop a language for a diverse set of machine architectures (Cray-2, the Manchester Prototype Data Flow Computer, multiprocessor VAX and HEP-1), we felt that an intermediate form would provide a common language for the implementors of SISAL. The reader may wish to consult the SISAL reference manual for a discussion of how IF1 fits into the overall plan of SISAL implementations.

IF1 has changed form dramatically since its first draft. The major change occurred at a meeting held at DEC in Hudson, Massachusetts in February, 1983. At that time the hierarchies of graphs were introduced, eliminating the need for explicit control lines and nodes. Since then we have eliminated the need for uniquely numbered nodes, and have placed extra type information in graphs to ensure type correctness when linking functions from different compilation units.

IF1 has not only been useful for decoupling the front-end of the compiler from the code generator, but it is the form read by many analysis programs. Machine-independent optimizations such as common-subexpression elimination and loop-invariant removal operate on graphs expressed in IF1. Machine-dependent analysis, such as finding vector operations or partitioning a graph over multiple processors will also be done using supersets of IF1.

The authors wish to thank the many people who have read intermediate drafts of this document. We especially thank Michael Welcome of LLNL for checking all of the examples and for helping produce the IF1 graphs on the Macintosh, and Robert Kim Yates for pointing out so many details that needed better explanation. We also thank the implementors of SISAL and IF1 for their patience as we strove to improve IF1 through frequent changes. It was their experience that suggested changes and improvements.

2. Graphs in IF1

IF1 is a language based on **acyclic** graphs. There are four components to a graph: *nodes*, *edges*, *types* and *graph boundaries*. Nodes denote operations, such as add or divide; edges represent values that are passed from node to node; and types can be attached to each edge or function. Graph boundaries surround groups of nodes and edges.

For example, a graph for the expression (a+b)/5 would be represented by:

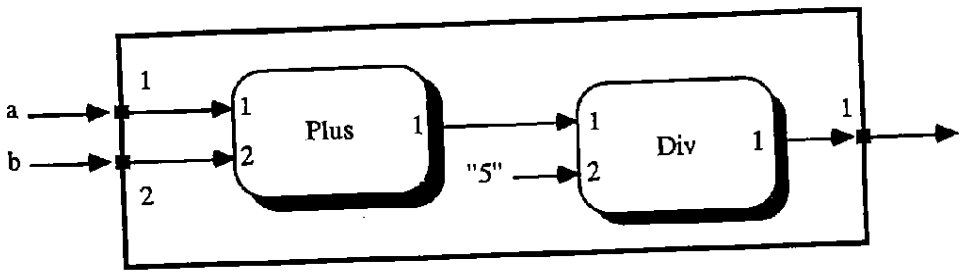


Figure 1. Graph of (a+b)/5

The outer (bold) box represents the graph boundary. This graph has two ports for input values (a and b), and one output port for the result.

The smaller boxes represent nodes. Over fifty nodes are defined in IF1 and a complete list of them is given in Appendix A. Both operations in the example take two input values and return one result, but the number of inputs and outputs vary according to the operation. For example, the node that builds an array can take any number of inputs, so that arrays of different sizes can be built. The number of inputs and outputs for each operation node is given in the description of that node (sections 6 and 7).

The numbers inside the nodes indicate ports, which are used to distinguish multiple inputs and outputs.

The arrows in the example denote edges. Edges represent data paths between nodes (or between nodes and graph boundaries). The edges in the example bring the values for a and b into the graph, pass their sum from the Plus node to the Div node, and send the result out of the graph. The edges also carry type information, which is not shown in the picture.

A special type of edge is used to describe literal constants. Literal edges do not contain a source node or port, but they contain the text of the literal as a string. The "5" in the example is a literal edge.

Each edge may carry type information. Strongly typed languages will place type information on each edge in the graph; untyped languages might not give any type information. Types can be specified for user-defined types as well as the

built-in types.

Comments may be used for any purpose, but they are the appropriate means for giving names to edges, or remembering the line that generated an edge or node. See the next section for more detail on comments.

3. The form of IF1

IF1 files comprise a number of *lines* that contain only printable ASCII characters, and are delimited by newline characters. The first non-blank (non-tab) character on the line distinguishes nodes from edges, etc.

C	comment
E	edge
L	literal edge
N, "{", "}"	node
T	type
G, X, I	graph

The rest of the line comprises fields that are delimited by white space (blanks or tabs). The number of fields in an IF1 line will depend on the type of line, and in some cases, on the input to a node. For example, an edge will always have five fields: source node, port on source node, destination node, port on destination node, and type. Any information beyond the expected number of fields is considered to be a comment.

Nodes and types contain *label* fields. IF1 uses integers as labels for nodes and types. The labels for nodes and types need not be disjoint, since we can tell by context whether a reference is to a node or a type. However, the "label scoping" rules for nodes and types differ.

Type labels exist only in one, global scope. Therefore all type labels within a single IF1 file must be unique (if multiple files are given to a program, the software that reads the files must merge the types sensibly).

Node labels need only be unique within a given graph. If a graph contains a subgraph, the node label definitions do not extend into the subgraph. The reason for this difference is that we feel that it is important to explicitly import and export values from graphs, while we see no harm in sharing types.

3.1. Comments

Comments can appear at the end of any line in an IF1 file, or as entire lines that begin with C (to win over the FORTRAN vote). Any printable characters (except newline) can appear in a comment, but certain conventions on the form of comments have been established.

3.1.1. Stamps

Comment lines that begin with C$ are called "stamps", and are used to mark an IF1 file with a record of some processing that has been done to it. As an example, an optimizer might stamp each file as it is processed. A single character follows the C$ to distinguish the stamps. Currently the following stamps are in use:

C$ A Array update analysis
C$ C Structure checker
C$ D Order nodes using data dependence
C$ E Common Subexpression elimination
C$ L Loop-invariant removal
C$ O Add offsets for use by the interpreter
C$ P Partitioning analysis
C$ S Stream analysis
C$ V Vector analysis

3.1.2. Pragmas

Pragmas appear at the end of lines in the comments field. They are used to record the source line that generated the edge, give a name to the edge, assign an offset to it, etc. The form is %aa=<anything>, where "aa" is a two character code, and <anything> is terminated by white space (a blank, tab, formfeed or newline). If more than one <anything> is needed, commas should be used as separators.

Currently, the following pragmas are generated by compilers:

%bd – bounds (subrange, takes two decimal numbers as arguments)
%na – name
%sl – source line
%op – op number within line of source

and these pragmas are the results of analysis:

%ar – size of activation record needed (for graphs)
%lz – edge carries a value that must be demanded
%mk – mark this edge "by reference" (%mk=r) or "by value" (%mk=v)
%of – offset in activation record
%st – style of memory allocation (%st=p for pointer, %st=c for contiguous)
%xy – position for node in graphic output (two decimal arguments)

The following table groups pragmas by the type of object that they modify.

IF1 Object	Code	Explanation	Example
Graph	ar	size of activation record (unitless)	%ar=75
Node	op	operation number within a line (unique within a source line)	%op=2
	sl	source line number of the operation	%sl=23
	xy	X-Y position for data flow graph layout	%xy=40,22
Edge	bd	subrange bounds (low, high inclusive)	%bd=0,255
	lz	value must be obained by demand	%lz=y or %lz=n
	mk	value or reference mark	%mk=v or %mk=r
	na	name (case is not important)	%na="root"
	sl	source line number where value appears	%sl=18
	st	memory allocation style (pointer or contiguous memory)	%st=p or %st=c

Table 1. Pragma Codes

We encourage others to use these pragma codes.

3.2. Nodes

Nodes can be either *simple* or *compound*. Simple nodes have the form

N label operation

where the operation is given by an ASCII string that represents an integer. A list of the operations and their associated integers is given in the BNF in Appendix A. Each operation is described in more detail in a later section.

Compound nodes contain subgraphs and span many lines. Their form is:

{
...
} label operation integer(s)

Again the operation is denoted by an ASCII string that represents an integer. The ellipsis represents an arbitrary number of lines for the subgraphs of the compound node. The number of integers at the end of the closing line depends on the number of subgraphs used in the compound node. An example of each compound node appears with its description in a later section.

3.3. IF1 Graph Boundaries

Graph boundaries begin with a line containing G and a type reference:

```
G 100
```

Graph boundaries that denote functions (as opposed to subgraphs of compound nodes) are denoted

```
G  <type reference>  "name"  for a local function definition
X  <type reference>  "name"  for a global function definition
I  <type reference>  "name"  for an imported function
```

The scope of node labels begins on the line following the G or X and extends until the occurrence of a record beginning with one of the following letters: G, X, I, or an unbalanced }. The label scope does not include the subgraphs of any compound nodes, which appear between *balanced* "{" and "}". The main purpose of the type field is to type graphs that are functions. Graphs that are subgraphs within compound nodes may choose to not give a type, which is denoted by a zero in the type field.

The graph boundary for imported functions merely associates a type descriptor with a function name. No nodes or edges can follow an imported function descriptor.

The graph boundary serves as a collector of imported and exported values for the edges inside it. The boundary is always implicitly labeled "0". When a value is needed from the graph boundary, the edge's source node has the label zero. Similarly, when a value is to be returned as the result of a graph, the value is sent to node zero.

Other values are sent and received from nodes numbered beginning at one. Each graph generally begins numbering its nodes at one and the node labels must be unique within the label scope of a graph.

3.4. Edges

Edges represent explicit data dependencies within the graph. They can give information about the type of the value that they represent, and they can also carry extra information in their comment fields. The extra information might give the name associated with the edge, or the source line that generated the edge.

Not all data dependencies are explicitly given in the edge list; some are implicit in the semantics of a compound node and its subgraphs. Such implicit connections are described in the semantics of each compound node.

Edges are 6-tuples: (source node, port, destination node, port, reference to type label, comment). If no type information is known, the reference should be label zero, meaning "no information provided".

Literals are 5-tuples: (destination node, port, reference to type label, string, comment). The string in a literal denotes the value for the literal; see the table below for their notation.

In the example of (a+b)/5 there are five edges:

source node	source port	destination node	destination port	type reference	string/comment
0	1	4	1	entry for integer	name is "a"
0	2	4	2	entry for integer	name is "b"
4	1	5	1	entry for integer	
		5	2	entry for integer	value is "5"
5	1	0	1	entry for integer	

If we assume that then integer type is labeled "3" we get the following IF1 description of the edges:

```
E   0   1   4   1   3   %na=a
E   0   2   4   2   3   %na=b
E   4   1   5   1   3
L               5   2   3   "5"
E   5   1   0   1   3
```

The IF1 code shows that "a" is imported by the graph on port 1, and that "b" is available on port 2. "5" is a literal that is used at node 5, port 2, and the result (a+b)/5 is exported by the graph on port 1.

In IF1, arbitrary fan-out of results is possible and is represented by multiple edges with the same source. However, *no fan-in is allowed*.

3.5. Literals

The notation for literals follows the form used in the SISAL language. Appendix B contains the BNF of literals in IF1, but we give an informal description here.

Function names are strings of alphanumeric characters. The case of the letters is unimportant.

Boolean values are either *T* or *F*.

Integer Values are strings of decimal digits.

Single-precision floating point values are either strings of decimal digits that contain a decimal point, or exponential notation with an E or e.

Double-precision floating point values use exponential notation with a D or d.

Character Values are single printable characters enclosed in apostrophes (single quotes), or are one of the printable representations of non-printing characters used in SISAL or "C" (e.g. '\b' '\n' '\007').

Null Values are denoted *nil*.

Error Values use the text of their representation in the high-level language.

Structured Values use the Fibre format [Skedzielewski, 1985] described tersely in Appendix B. We do not encourage the use of literals to denote structured objects; we prefer to use the array and record building nodes instead. The Fibre descriptions of complex structured objects tend to be lengthy, and not all implementations of IF1 will be able to process such long strings.

Type	Example
Function	"sqrt" "BoundaryCondition"
Boolean	"T" "F"
Integer	"137" "008"
Single	"2.71828" "3e8" ".0000863"
Double	"6.626196d-34" ".6022169D24"
Character	"'x'" "'\b'"
Null	"nil"

Table 2. Examples of Literals

3.6. Type Descriptors

Type descriptors are composed of entries for the basic types and entries for type constructors.

Types are organized as a collection of linked lists that tie constructors to base types and functions to arguments and results. Type entries provide type information about edges, whenever the compiler can produce them. Names can be attached to type entries by using the same convention as is used to name edges. Fields of records can be named in this way.

Entry	Code	Parameter1	Parameter2
Array	0	base type	
Basic	1	basic code	
Field	2	field type	next field
Function	3	argument type	result type
Multiple	4	base type	
Record	5	first field	
Stream	6	base type	
Tag	7	tag type	next tag
Tuple	8	type	next in tuple
Union	9	first tag	

Table 3. Type Entries

Type	Code	Type	Code
Boolean	0	integer	3
character	1	null	4
double	2	real	5

Table 4. Basic Type Codes

3.6.1. Basic Types

It is easiest to merely enumerate the IF1 for the basic SISAL types:

```
C   Label   Type code   Basic code   Comments (name)
T     1         1            0       %na=Boolean
T     2         1            1       %na=character
T     3         1            2       %na=double
T     4         1            3       %na=integer
T     5         1            4       %na=null
T     8         1            5       %na=real
```

The labels used here are not specifically assigned to these type entries in IF1, but the labels will be used for the basic types in all examples in this manual.

3.6.2. Array and Stream Types

Arrays and streams type entries merely point to the label of the base type of the array or stream.

C	Label	Type code	Base type	Comments (name)
T	61	6	1	stream of Booleans
T	76	0	6	array of reals
T	176	0	76	array of array of reals
T	99	6	9	stream of whatever type has label 9

3.6.3. Record types

Record type entries comprise a record header and one or more field entries. The fields contain a pointer to a type label for each field, and a pointer to the label of the next field entry. The last entry in the list contains a zero in the next field entry. The record header points to the first field.

C type Cartesian = record[X, Y: real]

C	Label	Type code		First Field	Comments (name)
T	1	5	21		%na=Cartesian
C	Label	Type code	Field Type	Next Field	Comments (name)
T	21	2	6	22	%na=X
T	22	2	6	0	%na=Y

3.6.4. Union Types

Union entries are similar to record entries, except that the position in the list corresponds to a tag, rather than a field.

C type IntList = union[Empty: null; Full: record[Mote: integer; Next: IntList]]

C	Label	Type code	Tag Type	Next Tag	Comments (name)
T	30	8		31	%na=IntList
T	31	7	5	32	%na=Empty
T	32	7	33	0	%na=Full
C	Label	Type code	Field Type	Next Field	Comments (name)
T	33	5		34	
T	34	2	2	35	%na=Mote
T	35	2	30	0	%na=Next

3.6.5. Function types

Functions comprise an argument list and a results list. Both lists are composed of Tuple entries. A function is defined by a graph whose graph boundary contains the name and type of the function.

The following lines are taken from an example at the end of the SISAL reference manual (the Sieve of Eratosthenes).

```
C type Si = stream [ integer ]
T 50    7          4              %na=Si        (7=Stream 4=label of base type)
```

A function declaration needs three pieces of information: an argument list (of tuples), a result list (of tuples), and a graph that has the name and type of the function.

```
C function Filter ( S : Si; M : integer returns Si)
C   Label   Type code     Args      Results
T    500         3         501        503
C   Label   Type code  Tuple Type    Next
T    501         8          50        502
T    502         8           4          0
T    503         8          50          0
C                                     Type
G                                      500     "Filter"
```

The labels used do not have to be contiguous numerals, nor even be monotonic. Further, the labels used for types and nodes can overlap.

The following is an example of how an external function would be represented in IF1:

```
C import Sqrt ( Q : real returns real )
T 51   3   52   52   (3=Function,52=Arg,52=Results)
T 52   8    4    0   (8=Tuple,4=Type,0=Next)
T  6   1    5        (1=Basic,5=Real)
I                51   "Sqrt" (51=function type)
```

The final example is a declaration drawn from the VAL reference manual (page 12):

```
C union [ THIS: array [ integer ] ; THAT, THE_OTHER : record [ C:real; D:boolean ] ]
T 100   8 101                                          (8=Union,101=FirstTag)
T 101   7 102 103        %na=THIS                      (7=Tag,102=Type,103=Next)
T 102   0   3
T 103   7 104 107        %na=THAT                      (7=Tag,104=Type,107=Next)
T 104   5 105
T 105   2   6 106        %na=C                         (2=Field,6=Type,106=Next)
T 106   2   1   0        %na=D                         (2=Field,1=Type,0=Next)
T   1   1   0            %na=Boolean                   (1=Basic,0=Boolean)
T 107   7 104   0        %na=THE_OTHER                 (7=Tag,104=Type,0=Next)
```

4. Functions

A function is represented by a graph containing the body of the function. In a language that does not import global values (e.g. SISAL) the imports of the graph will correspond to the arguments of the function. The results of the graph correspond to the function results.

The graph receives arguments to the function when it is called (see "Call" node) and returns a set of values as results. A function name and type are associated with a graph by following a G (graph) record at the outermost level with a type reference and a string (e.g. G 100 "sqrt"). If the function is to be exported from the current compilation unit the record begins with X, followed by the type reference and a string (e.g. X 100 "BoundaryConditions").

In order to provide type-checking across compilation units, IF1 defines a third form of function header. Lines beginning with I (imports), followed by a type reference and a string, associate types with the names of imported functions. However, no graph body can be present. See the description of the "sqrt" function below for an example of an imported function.

Type descriptors for functions contain two lists: an argument list and a result list. Each of the lists comprises list elements (tuples), which have a type reference and a pointer to the next list element. Pointers refer to type labels (which are integers) and the end of a list is denoted by a pointer to nil (zero).

An Example:

```
C import sqrt( X:real returns real)
C function Pythag ( x,y : real returns real )
C   sqrt( x*x + y*y )
C end function

T 7     Basic  5        %na=real
T 10    Function        11 13
T 11    Tuple  7        12
T 12    Tuple  7        0
T 13    Tuple  7        0

I                       20      "sqrt"
T 20    Function        13 13

G                       7       "Pythag"

N 1     Times
E       0 1     1 1     7       %na=x
E       0 1     1 2     7       %na=x
N 2     Times
E       0 2     2 1     7       %na=y
E       0 2     2 2     7       %na=y
N 3     Plus
E       1 1     3 1     7
E       2 1     3 2     7
N 4     Call
L               4 1     20      "sqrt"
E       3 1     4 2     7
E       4 1     0 1     7
```

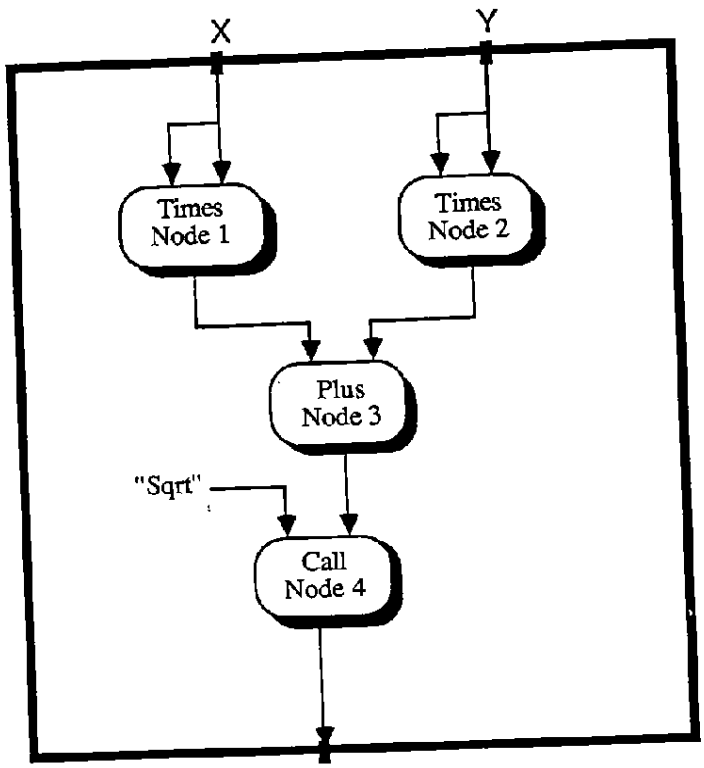


Figure 2. Example of a function graph

5. Nodes

Nodes represent operations on values. There are two types of nodes, *simple* and *compound*. Compound nodes contain subgraphs while simple nodes do not. The semantics of simple nodes describe the relation of the inputs of the node to its outputs. The semantics of compound nodes describe the ways in which the subgraphs of the compound node interact.

Most simple nodes have a fixed number of input and output ports although there are variable numbers of inputs to nodes such as those that build structured data types. Compound nodes gather the values needed by their subgraphs, so they generally have an arbitrary number of input and output ports. Any unconnected port is simply unused, or "grounded".

The compound nodes are hierarchically defined, so that substructures are denoted by subgraphs. Some examples of nodes are:

Node	Kind	Inputs	Outputs	Subgraphs
Plus	simple	2	1	0
AElement	simple	2	1	0
AScatter	simple	1	2	0
RBuild	simple	any	1	0
LoopB	compound	any	any	4
Select	compound	any	any	>2

The subgraphs of LoopB are *initialize*, *body*, *test*, *results*. The subgraphs of Select are a *selector* and two or more *alternatives*.

Simple nodes in an IF1 file are denoted by lines that begin with an 'N' and are followed by a numeral that represents the node. Lines beginning with '{' and '}' delimit a compound node and its subgraphs. The '}' is followed by a label, an operation code, and extra information that is specific to that kind of node. A complete list of nodes and their associated numerals is given in Appendix A.

5.1. Simple Nodes

Simple nodes are not necessarily uncomplicated, but they have no subnodes.

Many of the arithmetic operation nodes are generic. The type of the operation can be determined by examining the types of the input and output arcs. In the following descriptions, let

arith	= { integer, real, double }
algeb	= arith + { Boolean }
atom	= algeb + { char}
T	= any type

We will also use the following notation to construct cartesian products of types

S × T	cartesian product of values of type S and T
(T)*	zero or more occurrences of values of type T
(T)+	one or more occurrences of values of type T
[T]	zero or one occurrence of a value of type T

5.1.1. Arithmetic and Boolean Nodes

Plus
Times
Min
Max *algeb* × *algeb* → *algeb*

The types of both of the operands must be the same as the type of the result. When the domain is the set of Boolean values, Plus and Max accomplish the "Or" function, while Times and Min accomplish the "And" function.

Minus
Div
Mod *arith* × *arith* → *arith*

The types of both of the operands must be the same as the type of the result. For integers, *Div* (N1, N2) truncates the result towards zero.

We define the modulus of two numbers (N1, N2) to be:

min r ≥ 0 such that N1 = N2*k + r, k an integer

a	b	mod(a,b)	why
5	3	2	5 = 1*3 + 2
-5	3	1	-5 = -2*3 + 1
-5	-3	1	-5 = -2*(-3) + 1
5	-3	2	5 = -1*(-3) + 2

Table 5. Behavior of mod

Exp *arith* × *arith* → *arith*

Exp is the function that takes the first input to the power of the second input. [redacted]e (the latter only occurs when one of the operands is a double). Error conditions can occur based on the sign and type of the arguments to Exp. The following table describes the conditions leading to errors.

Type of Arg2					
Integer			Real or Double		
Arg1	Arg2	Result	Arg1	Arg2	Result
= 0 ≠ 0	≥ 0 any	$Arg1^{Arg2}$	=0 >0	≥ 0 any	$Arg1^{Arg2}$
= 0	< 0	*error*†	= 0 < 0	< 0 any	*error*†

Table 6. Error behavior of Exp

Abs
Neg *arith* → *arith*

Absolute value and Negation

Not *Boolean* → *Boolean*

Less
LessEqual
Equal
NotEqual *atom* × *atom* → *Boolean*

Both of the operands must be of the same type. *NotEqual* and *LessEqual* represent the Boolean functions exclusive-or and implication.

IsError *T* × *T* → *Boolean*

The first value is a literal whose string denotes one of the error values (e.g. "error", "undef", "pos_over"). IsError returns *true* if the second value is the same error value as the first. The special error value "error" matches **any** error value on the second port.

† *error* denotes the appropriate error of type real or double.

5.1.2. Miscellaneous Nodes

Bool *integer → Boolean*

Zero maps to *false* and one maps to *true*. All other integers map to the appropriate error value.

Char *integer → character*

The result is only defined on integers in the range 0 to 127. If the integer does not map into a valid ASCII character, the result is the appropriate error value of type Char.

Double *real → double*
integer → double

If the result is outside the range of double values, the result is an error of type Double

Int *atom → integer*

Real and Double values are rounded to the nearest integer by adding 0.5 and taking the floor of the result. If the result is outside the range of integer values, the result is an error value of type integer. *False* is mapped to zero, and *true* is mapped to one. Characters are mapped to their associated ASCII code.

Floor *real → integer*
double → integer

The value returned is the greatest integer less than or equal to the input value. If the result is outside the range of integer values, the result is an error of type integer.

Trunc *real → integer*
double → integer

The value returned is formed by truncating any fractional part of the input value, then converting it to an integer. If the result is outside the range of integer values, the result is an error value of type integer.

Single *double → real*
integer → real

If the result is outside the range of real values, the result is an error of type real.

NoOp *(T)+ → (T)+*

Each output is equal to its corresponding input.

5.1.3. Array and Stream Manipulation Nodes

Arrays and Streams have nearly the same set of operations available on them in IF1. Only the AAdjust, AReplace and ASetL operations are not allowed on streams. The use of Arrays and Streams in subgraphs of LoopA, LoopB and Forall may require the use of Scatters and Gathers, which are described in the section on multiple values.

ABuild
integer × *(T)** → *array(T)*
integer × *(T)** → *stream(T)*

Arrays and streams are built by providing a lower bound (ignored for streams) on port one of ABuild, and values on the following ports. Any number of values (including zero) may be present, but they must occur on contiguous ports (no gaps). The type of the array or stream built is found on the edge or edges leaving the ABuild node.

AFill
integer × *integer* × *T* → *array(T)*
integer × *integer* × *T* → *stream(T)*

AFill creates an array or stream filled with the value given on port three. The integer on port one gives the lower bound, and the integer on port two gives the upper bound of the array or stream being built. If the lower bound is higher than the upper bound an empty array or stream is created. If a stream is built, the integer on port one is ignored and the integer on port two provides the length of the stream.

AElement
array(T) × *integer* → *T*
stream(T) × *integer* → *T*

Element extracts the element of array or stream at the index position given on port two. Note that only one level of subscripting is done.

AReplace *array(T)* × *integer* × *(T)+* → *array(T)*

AReplace produces an array that differs from the input array at a given index. The integer on port two gives the first index position to change. If more than one value is given, they are placed at consecutive index positions.

ACatenate
array(T) × *(array(T))+* → *array(T)*
stream(T) × *(stream(T))+* → *stream(T)*

ACatenate produces the catenation of its input streams or arrays. The types of all input edges must be the same.

ALimL
ALimH
array(T) → *integer*
stream(T) → *integer*

Give the low (high) bounds of the stream or array. A stream will always have a lower bound of one.

ASize
APrefixSize
array(T) → *integer*
stream(T) → *integer*

ASize returns the number of elements *in the highest dimension* of the array or stream on port one.

APrefixSize returns the number of elements *in the highest dimension* that might be extracted from an array or stream.

If the array is in error, ASize and APrefixSize may differ. APrefixSize *never* yields an error value, while ASize may do so. See the section in the SISAL reference manual [McGraw 1985] on error handling for more detail.

AIsEmpty *array(T)* → *Boolean*
stream(T) → *Boolean*

AIsEmpty is true if and only if APrefixSize=0.

AAddL *array(T)* × *T* → *array(T)*
AAddH *stream(T)* × *T* → *stream(T)*

Add an element to the low (high) end of the array or stream.

ARemL *array(T)* → *array(T)*
ARemH *stream(T)* → *stream(T)*

Remove the low (high) element of the array, stream, or multiple. If the input stream or array is empty, the result is an error value of the same type as the output of the node.

ASetL *array(T)* × *integer* → *array(T)*

The lower bound of the array is shifted to the given integer value. All other indices are similarly shifted.

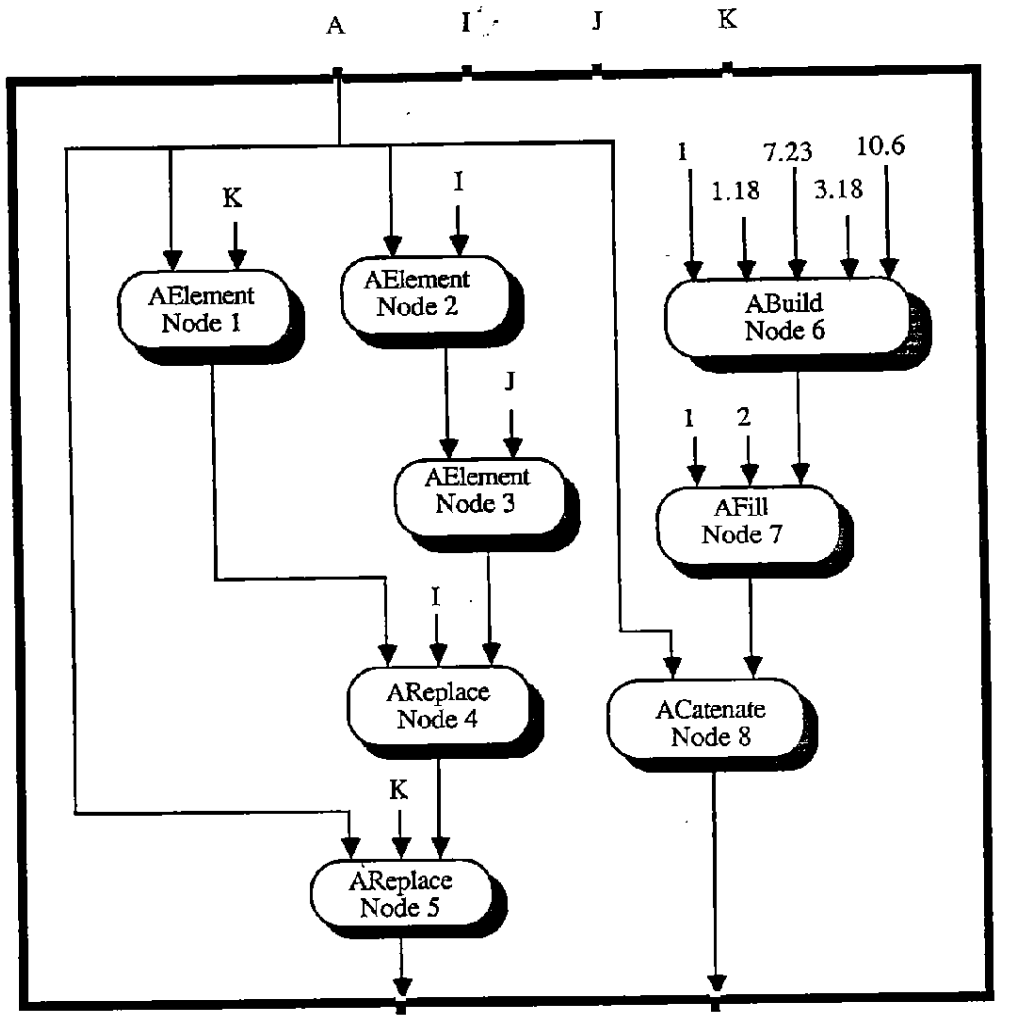
AExtract *array(T)* × *integer* × *integer* → *array(T)*

A subset of values of the input array are extracted and placed in the output array. The low index is given by the integer on port one, and the high index is given by the integer on port two. If the array is in error, error-valued arrays are produced. If the indices are out of bounds but the input array is not in error, a non-error array is produced, which has an error value for any element corresponding to an index that is out of bounds.

In the following figure some of the edges are not explicitly shown in order to make the graph easier to read. All of the inputs labelled I, J, or K are actually connected to ports 1, 2, or 3, respectively.

Examples of Array Manipulation Nodes

```
C type aareal = array[ array[ real ] ];
C
C function ArrayExample( I, J, K: integer; A: aareal returns aareal, aareal )
C
C  A[ K, I: A[ I, J ] ],
C  A || array_fill( 1, 2, array[ 1: 1.18, 7.23, 3.18, 10.6 ] )
C
C end function

T     4  Basic      integer
T     8  Basic      real
T    56  Array        6
T   156  Array       56
T   256  Function   261   257
T   257  Tuple        4   258
T   258  Tuple        4   259
T   259  Tuple        4   260
T   260  Tuple      156   261
T   261  Tuple      156     0

X                                256  "ArrayExample"
E       0 1          1 1     156  %na=A
E       0 4          1 2       4  %na=K
N    1  AElement
E       0 1          2 1     156  %na=A
E       0 2          2 2       4  %na=I
N    2  AElement
E       2 1          3 1      56  A[ I ]
E       0 3          3 2       4  %na=J
N    3  AElement
E       1 1          4 1      56  A[ K ]
E       0 2          4 2       4  %na=I
E       3 1          4 3       6  A[ I, J ]
N    4  AReplace
E       0 1          5 1     156  %na=A
E       0 4          5 2       4  %na=K
E       4 1          5 3      56  A[ I: A[ I, J ] ]
N    5  AReplace
L                    6 1       4  "1"
L                    6 2       6  "1.18"
L                    6 3       6  "7.23"
L                    6 4       6  "3.18"
L                    6 5       6  "10.6"
N    6  ABuild
L                    7 1       4  "1"
L                    7 2       4  "2"
E       6 1          7 3      56  array...
N    7  AFill
E       0 1          8 1     156  %na=A
E       7 1          8 2     156  array_fill( 1, 2, array... )
N    8  ACatenate
E       5 1          0 1     156  A[ K: A[ I: A[ I, J ] ] ]
E       8 1          0 2     156  A || array_fill( 1, 2, array... )
```



$$A=\begin{bmatrix} 1.1 & 2.2 & 3.3 & 4.4 \\ 2.0 & 3.0 & 4.0 & 5.0 \\ 0.4 & 0.3 & 0.2 & 0.1 \end{bmatrix}$$

I=1, J=3, K=2

Input to ArrayExample

$$\begin{bmatrix} 1.1 & 2.2 & 3.3 & 4.4 \\ \boxed{3.3} & 3.0 & 4.0 & 5.0 \\ 0.4 & 0.3 & 0.2 & 0.1 \end{bmatrix}$$

First result

$$\begin{bmatrix} 1.1 & 2.2 & 3.3 & 4.4 \\ 2.0 & 3.0 & 4.0 & 5.0 \\ 0.4 & 0.3 & 0.2 & 0.1 \\ \hdashline 1.18 & 7.23 & 3.18 & 10.6 \\ 1.18 & 7.23 & 3.18 & 10.6 \end{bmatrix}$$

Second Result

Figure 3. Array manipulation nodes

5.1.4. Record and Union Manipulation Nodes

Three operations are available on records: create, replace and extract all fields. We define only two operations involving objects of type Union: create and distinguish by tag (TagCase). The latter is a compound node and is discussed in another section.

RBuild $T_1 \times T_2 \times \ldots \times T_n \rightarrow record$

RBuild creates a record value with n fields. The correspondence between fields and input port numbers is determined by the ordering of fields in the type description. Port one corresponds to the first field, port two corresponds to the second field, etc. The record's type can be found by looking at the type of the arc leaving the node.

All ports corresponding to fields of a record must receive a value.

RBuild $T \rightarrow union$

The input ports of RBuild are associated with each tag of the Union type. The correspondence between port numbers and tags is determined by the ordering of the tags in the type description.

Only one port of an RBuild node that builds a union can receive a value. The tag is determined by the position of the input port in use. Port one corresponds to the first tag, port two corresponds to the second tag, etc.

RElements $record \rightarrow (T)+$

The values for each field of the input record are available at the output ports of RElements. The correspondence between port numbers and fields is determined by the ordering of fields in the type description, as in RBuild.

RReplace $record \times (T)+ \rightarrow record$

RReplace has a port for the input record and one for each field in that record. The first port of the RReplace receives the record that supplies the basis for the new record. Ports two through (number of fields + 1) give value(s) to be substituted in the corresponding field. Any number of ports may receive new values; empty input ports denote fields that are unchanged.

RReplace is merely a shorthand notation when replacing one or more fields of a record. The same effect can be accomplished by a RElements node immediately followed by a RBuild node, where the unchanged values are passed from the outputs of RElements to the corresponding inputs of RBuild.

Example of Record Manipulating Nodes

```
C   type Cart    = record[ X,Y: real ]
C
C   Origin       := record[ X: 4.2; Y: 3.1 ];
C   XX           := Origin.X * 2.0;
C   Home         := Origin replace[ Y: XX ];

T    6  Basic      Real
T   25  Record     26          %na=Cart
T   26  Tuple      6 27        %na=X
T   27  Tuple      6 0         %na=Y
G    0
L                  1 1     6   "4.2"
L                  1 2     6   "3.1"
N    1  RBuild
E       1 1        2 1     6   %na=Origin
N    2  RElements
E       2 1        3 1     6
L                  3 2     6   "2.0"
N    3  Times
E       1 1        4 1    25
E       3 1        4 3     6   %na=XX
N    4  RReplace
E       4 1        0 1    25   %na=Home
```

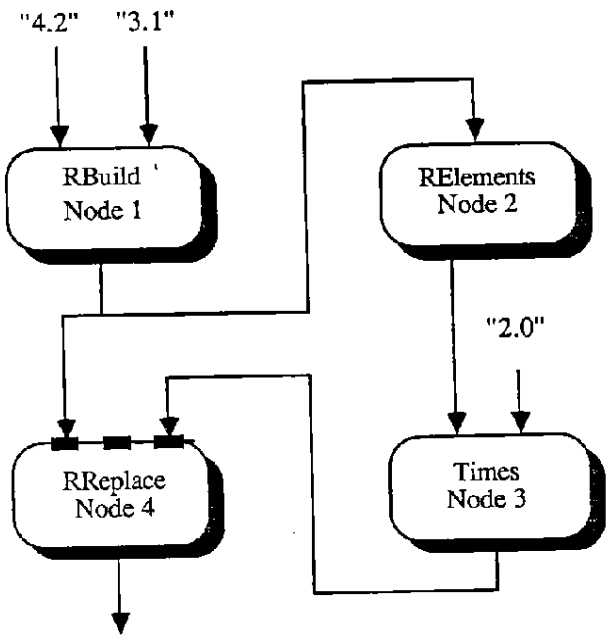


Figure 4. Record Operations

Example of Building a Union Object

```
C type CharTree = Union[   Leaf: character;
C                          Node: record[ L, R: CharTree ]
C                          ]

T    2   Basic     character
T   10   Union     11                %na=CharTree
T   11   Tag        2 12             %na=Leaf
T   12   Tag       13  0             %na=Node
T   13   Record    14
T   14   Field     10 15             %na=L
T   15   Field     10  0             %na=R

C function BuildTree( returns CharTree )
C let
C           LeafC    := union CharTree[ Leaf: 'C' ];
C           LeafI    := union CharTree[ Leaf: 'I' ];
C           LeafT    := union CharTree[ Leaf: 'T' ];
C           Node1    := union CharTree[ Node: record[ L: LeafC; R: LeafI ] ];
C in
C           union CharTree[ Node: Record[ L: InternalNode; R: LeafT ] ]
C end let

X                                20   "BuildTree"
T   20   Function    21
T   21   Tuple       10    0

L                     1 1     2    "'C'"
N    1   RBuild
L                     2 1     2    "'I'"
N    2   RBuild
L                     3 1     2    "'T'"
N    3   RBuild
E        1 1          4 1    10    %na=LeafC
E        2 1          4 2    10    %na=LeafI
N    4   RBuild
E        4 1          5 2    13    %na=Node1
N    5   RBuild
E        5 1          6 1    10
E        3 1          6 2    10    %na=LeafT
N    6   RBuild
E        6 1          7 2    13
N    7   RBuild
E        7 1          0 1    10    %na=List
```

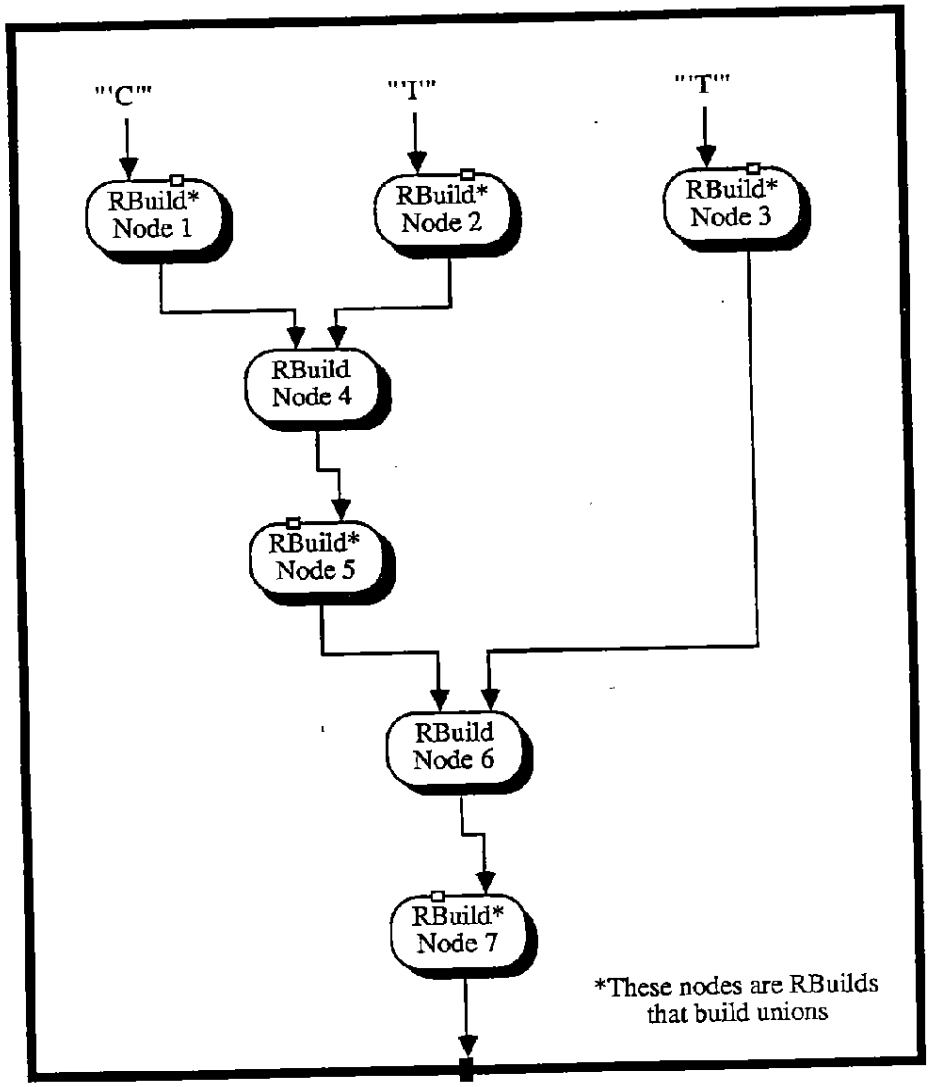


Figure 5. Example of Union Building Nodes

5.1.5. Nodes Dealing with Multiple Values

An arc with type Multiple(T) represents a sequence of values of type T.

In IF1 we need a way to transmit all of the loop values into *Forall* bodies from the AScatter and RangeGenerate nodes. Similarly, we must pass a sequence of values out of all *Forall*, *LoopA and LoopB* bodies for use in the test and returns sections. The type constructor *multiple* denotes such values, and the nodes below operate upon them.

Multiple values have index positions associated with them. The indices are available for use as values in the loop bodies. They are also used to place an ordering on the values that are built into streams or arrays (AGather node), and to place values in the correct order for reduction operations.

The next two nodes can only appear in the generator subgraph of Forall nodes.

AScatter *array(T)* → *multiple(T)* × *multiple(integer)*
stream(T) → *multiple(T)* × *multiple(integer)*

The elements of the input are placed sequentially on port one in the same order as they occur in the array or stream. Their corresponding index values are placed on port two.

RangeGenerate *integer* × *integer* → *multiple(integer)*

A sequence of integers is placed on output port one. The (inclusive) range is given on input ports one and two. If the value on port one is greater than the value on port two, the output value is a null multiple.

The rest of the nodes in this section can only appear in the returns subgraph of a Forall, LoopA or LoopB node.

AGather *integer* × *multiple(T)* [× *multiple(Boolean)*] → *array(T)*
integer × *multiple(T)* [× *multiple(Boolean)*] → *stream(T)*

An array or stream is built from the values on the second port (multiple(T)). Port one (integer) gives the lower bound of the array; it is ignored for streams. An edge on port three (multiple(Boolean)) determines whether or not to include the value in the array or stream. A value is included if its corresponding Boolean is true, or if the third port is unused.

Reduce
RedLeft
RedRight
RedTree *function* × *T* × *multiple(T)* [× *multiple(Boolean)*] → *T*

The Reduce nodes take a binary function on port one, with unit value on port two (type T), and applies it to the multiple values on port three (multiple(T)). If the optional edge on port four (multiple(Boolean)) is present, only those values on port three whose corresponding Boolean is "true" will be used in the reduction. Otherwise, all values on port three participate.

The function must be of type (T × T → T); it is denoted by a literal on port one. The string value of that literal is one of { "sum", "product", "least", "greatest", "catenate"}. The associativity of the operation is either unspecified (*Reduce*), left (*RedLeft*), right (*RedRight*), or pairwise recursive (*RedTree*). The unit value for each function is given by the table below.

Function	Type	Unit Value
sum	integer × integer → integer	0
	real × real → real	0.0
	double × double → double	0D0
	Boolean × Boolean → Boolean	false
product	integer × integer → integer	1
	real × real → real	1.0
	double × double → double	1D0
	Boolean × Boolean → Boolean	true
least	integer × integer → integer	+∞
	real × real → real	+∞
	double × double → double	+∞
	Boolean × Boolean → Boolean	true
greatest	integer × integer → integer	-∞
	real × real → real	-∞
	double × double → double	-∞
	Boolean × Boolean → Boolean	false
catenate	array(T) × array(T) → array(T)	array T[ ]

Table 7. Reduction Operators

The *least* and *greatest* reductions will map +∞ and -∞ to appropriate error values of the appropriate type.

FirstValue
FinalValue *multiple(T)* [× *multiple(Boolean)*] → *T*

Return the first or last value of the multiple on port one. If the optional multiple(Boolean) is present on port one, return the first or last value that occurs when its corresponding Boolean value is true.

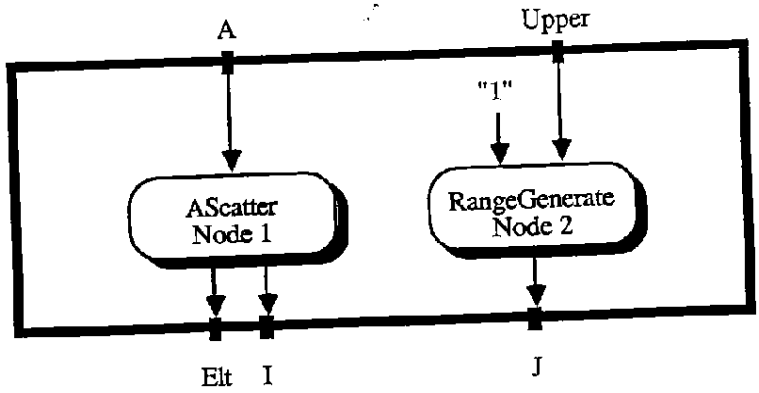
FirstValue is only used to provide the lower bounds for AGather nodes. FinalValue is used to produce the "value of X" in SISAL.

AllButLastValue *multiple(T)* [× *multiple(Boolean)*] → *multiple(T)*

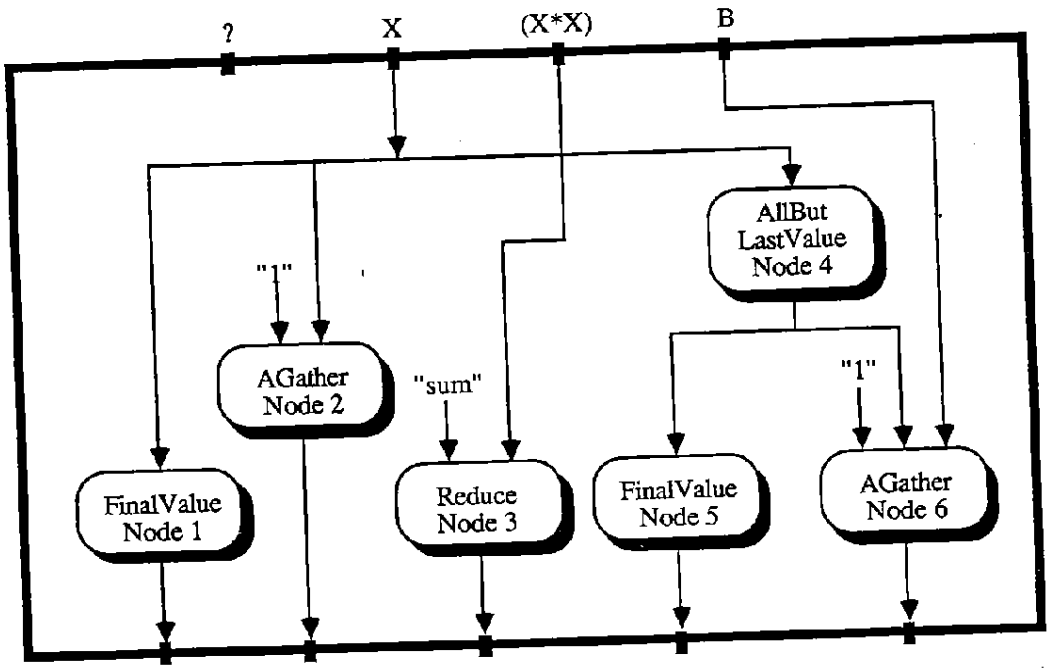
AllButLastValue is used on multiple values to remove the last value, in order to implement the return of **old** values in SISAL. It filters the multiple on port one by removing the last value from the multiple. If the optional multiple(Boolean) value is present on port two, remove the last value of the multiple that occurs when the Boolean is true. If the input multiple is empty, or none of the values in the multiple(Boolean) is true, the result of AllButLastValue is a multiple containing one error value of type T.

Examples of Multiple Value Nodes

```
T  2   Basic       Boolean
T  4   Basic       integer
T  6   Basic       real
T 60   Multiple    6                 assume that X is real
T 61   Array       6
T 62   Multiple    2                 and that B is Boolean
T 63   Function    64 65
T 64   Tuple       6 65              first arg is real
T 65   Tuple       6 0               second arg and result are real
T 66   Multiple    4

C      for Elt in A at I dot J in 1,Upper

G  0                                 generator subgraph
N  1   AScatter
E      0 1         1 1      61       %na=A
E      1 1         0 1      60       %na=Elt
E      1 2         0 2      66       %na=I
N  2   RangeGenerate
L                  2 1       4       "1"
E      0 2         2 2       4       %na=Upper
E      2 1         0 3      66       %na=J

C      returns
C        value of X,
C        array of X,
C        sum of X*X
C        old value of X,
C        old array of X when B
C      end for

G  0                                 returns subgraph
E      0 2         1 1      60       %na=X assume port 2
N  1   FinalValue
E      1 1         0 1       6       first result
L                  2 1       4       "1"
E      0 2         2 2      60       %na=X
N  2   AGather
E      2 1         0 2      61       second result
L                  3 1      63       "sum"
E      0 3         3 2      60       contains X*X
N  3   Reduce
E      3 1         0 3       6       third result
E      0 2         4 1      61       %na=X
N  4   AllButLastValue
E      4 1         5 1      60
N  5   FinalValue
E      5 1         0 4       6       fourth result
L                  6 1       4       "1"
E      4 1         6 2      60
E      0 4         6 3      62       %na=B assume port 4
N  6   AGather
E      6 1         0 5      61       fifth result
```



Multiple value nodes in the Generator subgraph



Multiple value nodes in the Returns subgraph

Figure 6. Example of Nodes that Deal with Multiples

5.1.6. Nodes Dealing With Functions

Call *function* × *(T)** → *(T)+*

The function given on port one is given the arguments (if any) on ports two and up. The order of the values is the same as the order of the arguments in the type table entry for the function. The results of the function application are returned on the outputs of the call node.

The function can be any object of type *function*, but it will be a literal value in both VAL and SISAL. The literal's type reference field will point to an entry for a function in the type table; the types of the arguments and results must match the types of the inputs and outputs of the Call node. The name of the function is given in the value string of the literal.

BindArguments *function* × *(T)** → *function*

BindArguments returns a new function whose arguments are bound to corresponding values on the input ports of the node. If no value is present on an input port, then that parameter is still a bound variable in the resulting function. The order of the ports is the same as the order of the arguments in the type entry for the input function.

5.2. Compound Nodes

Compound Nodes contain subgraphs. The number of subgraphs may be fixed (in Forall, LoopA and LoopB nodes), or may vary (in Select and TagCase node). The semantics of each compound node relate its inputs and outputs to the inputs and outputs of its subgraphs.

The representation of a compound node in IF1 is:

```
{
G 0
...          subgraph zero
G 0
...          subgraph one
.
.
.
G 0
...          subgraph n
} label OperationCode  k  a1 a2 ... ak
```

The first line merely marks the beginning of a compound node. Each subgraph begins with a G record (followed by optional comments), and terminates when either another G record, or an unmatched } is found. Remember that the zero after the G signifies "no type given". The last line closes the compound nodes and gives a node label, an integer code for the compound node and a list of integers (k a_1 a_2 ... a_k) called an *association list*.

A list of compound nodes and their associated integers is given in Appendix A.

Association List The association list links the subgraphs that are described in the semantics of each compound node with the subgraphs in the IF1 file. The subgraphs of a compound node may appear in any order in the text. The association list defines a mapping between the textual ordering of the subgraphs and their intended use by the compound node. Since each node uses the subgraphs differently, the association list has different meanings for each kind of compound node.

For example, the LoopA node has four subgraphs: Initialization, Test, Body, Result. The association list for the LoopA has the form

4 <init> <test> <body> <results>

where the last four fields are filled in with integers that point to the LoopA's subgraphs. Here is a skeleton for the IF1 code that might be produced from a SISAL **for** expression:

```
{
G 0              init
... (subgraph zero)
G 0              body
... (subgraph one)
G 0              test
... (subgraph two)
G 0              results
... (subgraph three)
} 100 LoopA  4 0 2 1 3
```

This association list connects subgraph zero with the initialization, subgraph two with the test, subgraph one with the body, and subgraph three with the results. The reason that the body graph might precede the test in a LoopA node is that the textual form of such a loop in SISAL places the expression for the body textually before the expression for the test.

Another reason to use association lists is to allow a subgraph to be shared, rather than copied. The only likely use of sharing occurs in the TagCase node, where several tags may share the same expression.

Edges in a subgraph use node zero to identify both the inputs and the outputs of the subgraph. The inputs to the subgraph have their source at node zero, and their results use node zero as their destination. We will call the values that cross graph boundaries ***imports*** and ***results***, and not refer to whether they are inputs or outputs of node zero itself.

The use of node zero to collect both the inputs and outputs of a subgraph causes an apparent data circularity in the subgraph. However, the inputs of a graph boundary are NOT connected (in the data flow sense) to the outputs of the boundary. In the figures, the graph boundary will surround the nodes and edges of a subgraph.

Implicit Dependence

Three kinds of implicit dependence can exist within a compound node:

(1) Data dependence between the compound node and its subgraphs
Type (1) dependence occurs when a compound node passes some of its input values to its subgraphs, or when a subgraph returns values that become the results of the compound node.

(2) Data dependence among subgraphs
Type (2) dependence usually appears in loops, where the loop values calculated in the initialization and loop body are passed to the results part and back into the loop body.

(3) Control dependence
Type (3) dependence occurs in selection and iterative loops, where a predicate determines which expression to evaluate or when an iterative loop is to terminate.

Classes of Values and Ports

When we discuss values that are passed to subgraphs of compound nodes we will speak of *classes* of values, such as *imported values* or *loop values*. Members of each class of values will be assigned to contiguous port numbers on the subgraph boundaries. Thus we can define a *port class* for each port on a subgraph boundary. Each compound node will define the ordering of port classes on its subgraph boundaries. For example, the ordering of ports on the input subgraph boundary to a loop body is <imported values> <loop values>. The port assignment is described in terms of the number of values in each class, denoted n_{class}.

Two tables of port usage will be given for each compound node. The first will assign port numbers to each class. The second will tell which ports can be used as imports and results of each subgraph.

Port classes are important in analysis routines, since they allow quick identification of loop values and loop invariants.

5.2.1. Select

SELECT
(value)+ → (value)+
N+1 subgraphs (Selector, N Alternatives)

The Select node is used to implement a multiway selection. Only a two-way selection exists in SISAL (viz. "if-then-else") but the Select node could be used to represent "case" expressions in other languages. There are N+1 subnodes: the selector (or predicate) and the alternatives. The selector returns exactly one value between zero and N-1 that determines which subgraphs results are to be used. Boolean selectors can be mapped to integers by the Int node, which maps false to zero and true to one.

The association list on the closing line of the compound node identifies the selector and the alternatives. The count of subgraphs must be at least three, since the select needs a selector and at least two alternatives. The first element of the association list identifies the selector subgraph. The following integers give the subgraph number to use for selector values 0, 1, ... count-2, respectively.

Implicit dependencies

(1) All input ports to the Select node are connected to corresponding input ports of all of its subgraphs (class K).

(2) The result ports of each Alternative subgraph are similarly connected to the corresponding output ports of the Select node (class R).

Class	Usage	Port Range
K	values imported by the compound node	1 .. n_K
R	result values	1 .. n_R
S	selector value	1 .. 1

Table 8. Port Assignments for Select

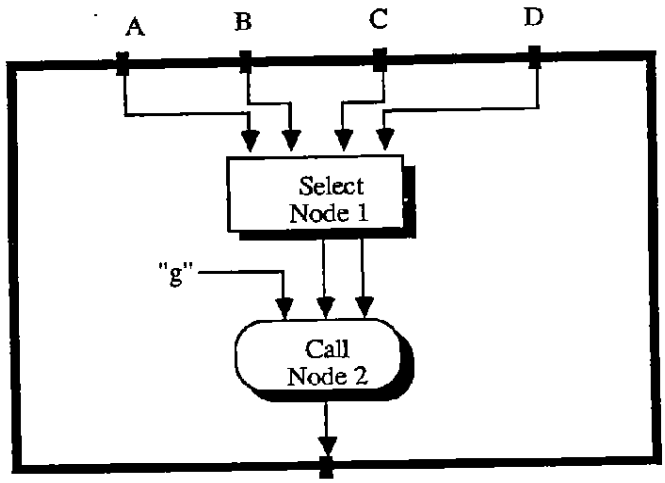
	Imports	Results
Select	K	R
selector	K	S
alternative	K	R

Table 9. Port Usage for Select

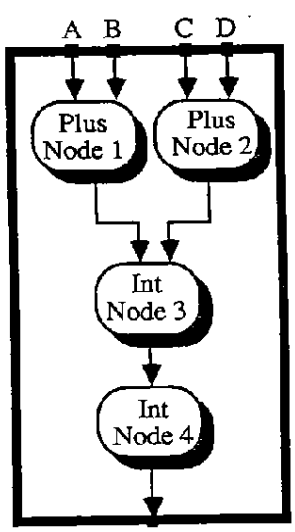
An example:

```
C import g( I, J : integer returns integer )
C function f( A, B, C, D : integer returns integer )
C  let X, Y :=
C    if A+B < C+D then
C      A, B
C    else
C      C, D
C    end if
C  in  g( X, Y )
C  end let
C end function % f

T  1     Basic     0     %na=Boolean
T  4     Basic     3     %na=Integer
T 10     Function  11  14
T 11     Tuple     4   12
T 12     Tuple     4   13
T 13     Tuple     4   14
T 14     Tuple     4    0
T 15     Tuple    13   14

I 15     "g"
X 10     "f"
{   n_k = 4 (A, B, C, D)

 G    0       Predicate (subgraph 0)
 N  1   Plus
 E      0   1    1   1    4       %na=A
 E      0   2    1   2    4       %na=B
 N  2   Plus
 E      0   3    2   1    4       %na=C
 E      0   4    2   2    4       %na=D
 N  3   Less
 E      1   1    3   1    4
 E      2   1    3   2    4
 N  4   Int
 E      3   1    4   1    Boolean
 E      4   1    0   1    4

 G    0       then part (subgraph 1)
 E     0   1    0   1    4       %na=A
 E     0   2    0   2    4       %na=B

 G    0       else part (subgraph 2)
 E     0   3    0   1    4       %na=C
 E     0   4    0   2    4       %na=D
} 1   Select     3   0 2 1        see comment below

C We must connect the false part (subgraph 2) to selector value 0 ("false")
C and the true part (subgraph 1) to selector value 1 ("true")

E      0 1    1 1   4   %na=A
E      0 2    1 2   4   %na=B
```

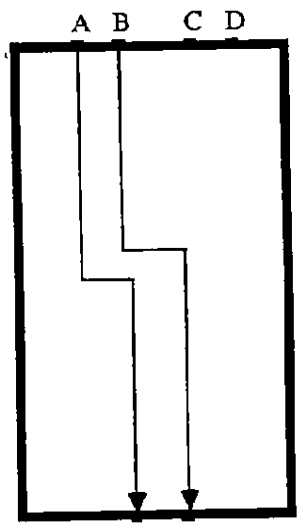
```
E     0 3    1 3   4   %na=C
E     0 4    1 4   4   %na=D

N 2   Call
L            2 1   15  "g"
E     1 1    2 2   4   %na=X
E     1 2    2 3   4   %na=Y
E     2 1    0 1   4
```



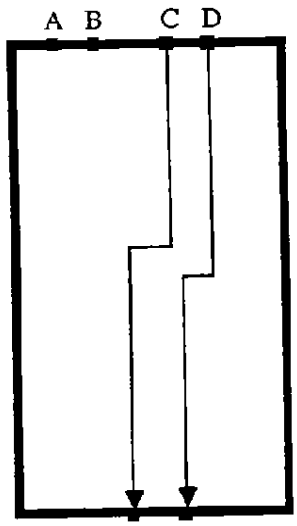
Graph of Function F



Select subgraph 0
Selector Graph



Select subgraph 1
True Branch

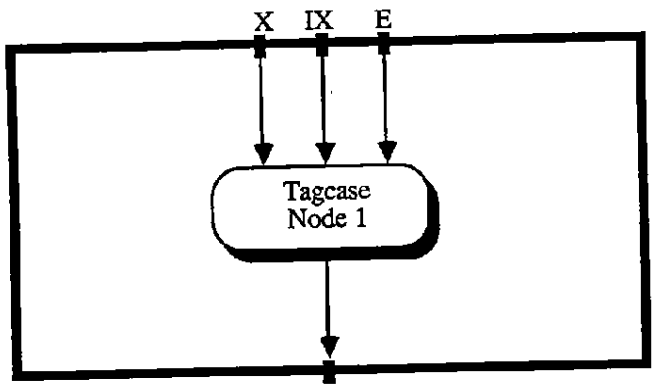


Select subgraph 2
False Branch

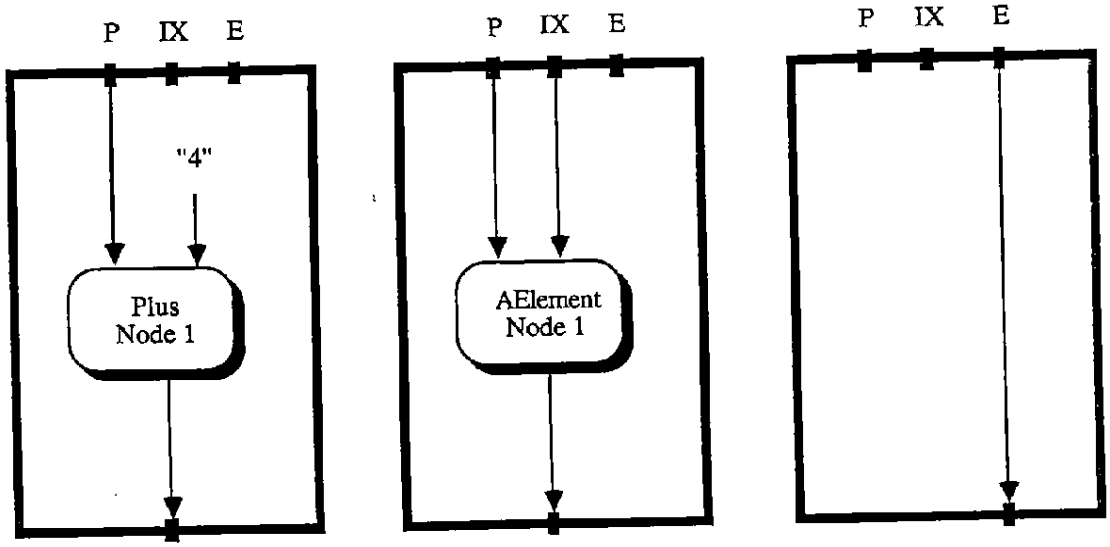
Figure 7. Example of Select

An Example:

```
C (let X be of type union [ A : integer;
C                           B : array[ integer ];
C                           C : real;
C                           D : Boolean ] )
C tagcase P := X
C  tag A: P + 4
C  tag B: P[IX]
C  otherwise: E
C end tagcase

T 1     Basic    0    %na=Boolean
T 4     Basic    3    %na=Integer
T 6     Basic    5    %na=Real
T 9     Array    4
T 10    Tag      1    0     %na=D
T 11    Tag      6    10    %na=C
T 12    Tag      9    11    %na=B
T 13    Tag      4    12    %na=A
T 14    Union    13

{    Compound 1 2

 G    0      subgraph 0
 N 1   Plus
 E     0  1   1  1   4     %na=P
 L           1  2   4     "4"
 E     1  1   0  1   4

 G    0      subgraph 1
 N 1   AElement
 E     0  1   1  1   9        %na=P
 E     0  2   1  2   4     %na=IX
 E     1  1   0  1   4

 G    0      subgraph 2
 E     0  3   0  1   4     %na=E
} 1   TagCase    4 0 1 2 2
```



Graph of Function Example



Tagcase subgraph 0

Tagcase subgraph 1

Tagcase subgraph 2

Figure 8. Example of Tagcase

5.2.3. Forall

FORALL
(value)+ → (value)+
3 subgraphs (Generator, Body, Results)

The Forall node is used to denote independent (as opposed to iterative) execution of multiple instances of an expression. It has three subgraphs: the generator, body and results.

The body contains the expression to be evaluated. The generator produces values for each instance of the body. It does so by producing at least one multiple value on ports in class "M". Each element of the multiple value is sent to a distinct instance of the body; they are not broadcast to all instances of the body. The values and instances of the body are *ordered* and the results subgraph will gather the results in the same order. When more than one multiple value is produced by the generator, the first value on *each* class M edge is send to the first instance of the body, and so on.

Implicit dependencies

(1) All inputs to the Forall node are available to each subgraph on the class K ports.

(2) The body subgraph receives one value from each class M port.

(3) The results of the body (class T) are connected to the inputs of the results subgraph.

(4) The results of the returns subgraph (class R) are connected to the output ports of the Forall node.

Cross products are represented by nested Foralls. Dot products are represented by more than one AScatter or RangeGenerate node in the generator subgraph.

Class	Usage	Port Range
K	values imported into the Forall node	$1 .. n_K$
M	multiple values	$n_K+1 .. n_K+n_M$
T	results of each Body	$n_K+n_M+1 .. n_K+n_M+n_T$
R	results of the Forall node	$1 .. n_R$

Table 12. Port Assignments for Forall

	Imports	Results
Forall	K	R
Generate	K	M
Body	K,M	T
Returns	K,M,T	R

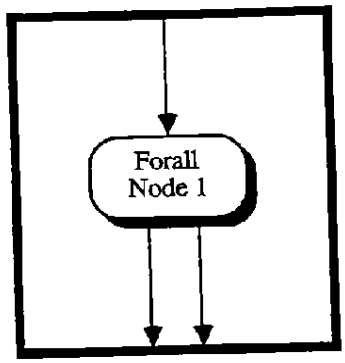
Table 13. Port Usage for Forall

An Example:

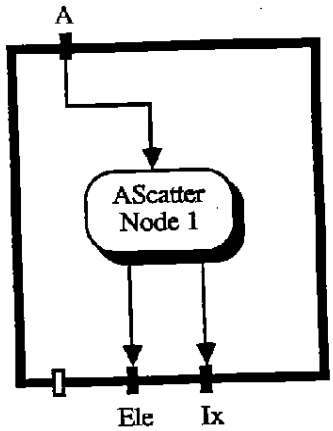
```
C type RealArray = array[ real ]
C
C global Sqrt( real returns real )
C
C function Example( S : RealArray returns RealArray, real )
C  for Ele in A at index IX
C    Sqr := Ele * Ele;
C    Root := Sqrt( Ele )/ real( array_size( A ) )
C  returns
C    array of Sqr
C    value of tree sum Root
C  end for
C end function

T 4    Basic        3          %na=Integer
T 6    Basic        5          %na=Real
T 7    Multiple     1
T 8    Multiple     4
T 9    Multiple     6
T 10   Array        6          %na=RealArray
T 12   Tuple        6     0
T 13   Tuple        6     12
T 14   Function     12    12   %na=Sqrt
T 15   Function     13    12   %na=Example

I                   12    "Sqrt"

G                   13    "Example"
{  n_K=1 (A)  n_M=2 (Ele, IX)  n_T=2 (Sqr, Root)

G                   0     generator (subgraph 0)
N 1    AScatter
E      0 1   1 1   10    %na=A
E      1 1   0 2   9     %na=Ele
E      1 2   0 3   8     %na=IX

G                   0     body (subgraph 1)
N 1    Times
E      0 2   1 1   6     %na=Ele
E      0 2   1 2   6     %na=Ele
N 2    Call
L             2 1   14    "Sqrt"
E      0 2   2 2   6     %na=Ele
N 3    ASize
E      0 1   3 1   10    %na=A
N 4    Single
E      3 1   4 1   4
N 5    Div
E      2 1   5 1   6
E      4 1   5 2   6
E      5 1   0 5   6     %na=Root
E      1 1   0 4   6     %na=Sqr
```

Forall

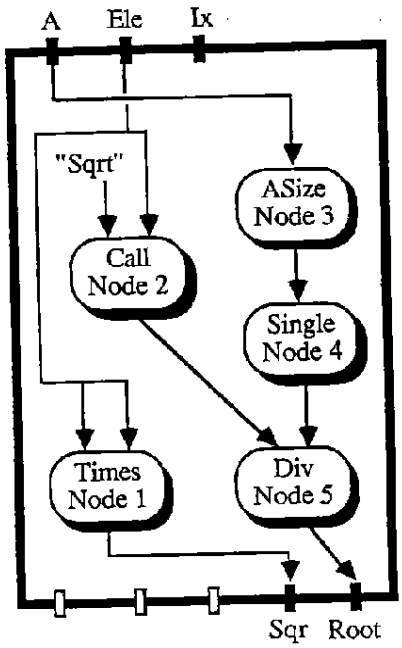
```
G                          0      returns (subgraph 2)
N 1   FirstValue
E     0 3   1 1   8
N 2   ReduceTree
L           2 1   15     "sum"
L           2 2   6      "0.0"
E     0 5   2 3   9      %na=Root
N 3   AGather
E     1 1   3 1   4
E     0 4   3 2   9      %na=Sqr
E     3 1   0 1   10
E     2 1   0 2   6

} 1   Forall 3  0 1 2
E     1 1   0 1   10
E     1 2   0 2   6
```



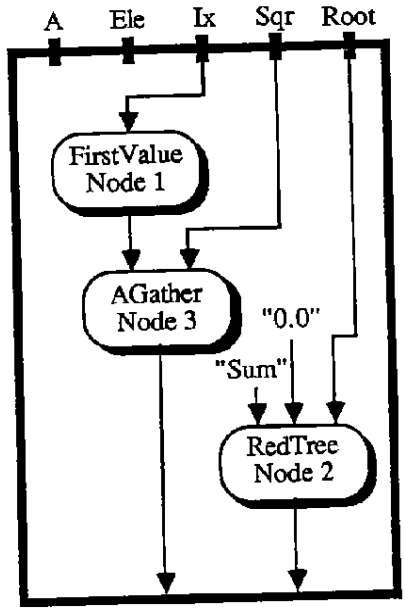
Graph of Function Example



Forall subgraph 0. Generator



Forall subgraph 1. Body



Forall subgraph 2. Returns

Figure 9. Example of Forall

5.2.4. LoopA

LOOPA
(value)+ → (value)+
4 subgraphs (Initialization, Test, Body, Returns)

LoopA is one of two iteration constructs (LoopB is the other). It repeatedly applies the loop body to a set of loop values, stopping and returning results when the test subgraph returns false. LoopA is the iteration compound node that tests for termination *after* the body has executed once. It has four subgraphs: initialization, test, body and returns. The association list's count field will contain four, and the four integers that follow will give the number of the subgraph corresponding to Initialization, Test, Body and Returns, in that order.

If the predicate should return an error value, the objects returned by the returns subgraph must be in error also.

Implicit dependencies

(1) All inputs to the LoopA node are available to each subgraph on the class K ports.

(2) The loop values from the output of initialization are connected to the input of the body and returns subgraphs on the class L ports.

(3) The loop values from the output of the body are connected to the inputs of the test, body and returns subgraphs on the class L ports.

(4) The derived loop values from the output of the body are connected to the inputs of the test, body and returns subgraphs on the class T ports.

(5) The results of the returns subgraph are connected to the output ports of the LoopA node on the class R ports.

(6) When the result of the test subgraph is false, the LoopA makes its results available at its output ports.

Class	Usage	Port Range
K	values imported into the LoopA node	$1 .. n_K$
L	loop values	$n_K+1 .. n_K+n_L$
T	local values generated in the loop body	$n_K+n_L+1 .. n_K+n_L+n_T$
B	boolean value (terminate when false)	$1 .. 1$
R	results of the LoopA node	$1 .. n_R$

Table 14. Port Assignments for LoopA

	Imports	Results
LoopA	K	R
Init	K	L
Test	K,L,T	B
Body	K,L	K,L,T
Returns	K,L	R

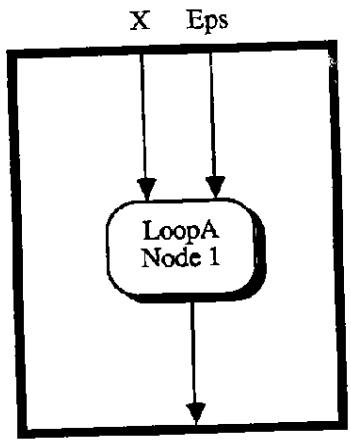
Table 15. Port Usage for LoopA

An Example:

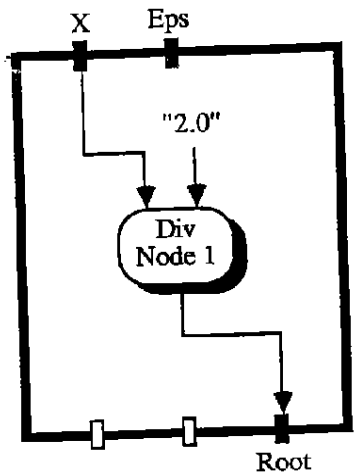
```
T 1   Basic      0     %na=Boolean
T 2   Basic      1     %na=Character
T 3   Basic      2     %na=Double
T 4   Basic      3     %na=Integer
T 5   Basic      4     %na=Null
T 6   Basic      5     %na=Real
T 7   Multiple   1
T 8   Multiple   4
T 9   Multiple   6

C function Sqrt( X, Eps : real returns real )
C  for initial
C    Root := X/2.0
C  repeat
C    Root := ( X /old Root + old Root ) / 2.0
C  until abs(X - (Root*Root) ) < Eps
C  returns value of Root
C  end for
C end function

T 10  Tuple      6     0
T 11  Tuple      6     1
T 12  Function   11    10

X                12    "SQRT"
{ n_K=2 (X, Eps) n_L=1 (Root) n_T=0


G                      0     Init (subgraph 0)
N 1   Div
E     0 1   1 1        6     %na=X
L           1 2        6     "2.0"
E     1 1   0 3        6     %na=Root

G                      0     Body (subgraph 1)
N 1   Div
E     0 1   1 1        6     %na=X
E     0 3   1 2        6     %na=Root
N 2   Plus
E     1 1   2 1        6
E     0 3   2 2        6     %na=Root
N 3   Div
E     2 1   3 1        6
L           3 2        6     "2.0"
E     3 1   0 3        6     %na=Root

G                      0     Test (subgraph 2)
N 1   Times
E     0 3   1 1        6     %na=Root
E     0 3   1 2        6     %na=Root
N 2   Minus
E     0 1   2 1        6     %na=X
E     1 1   2 2        6
```

LoopA

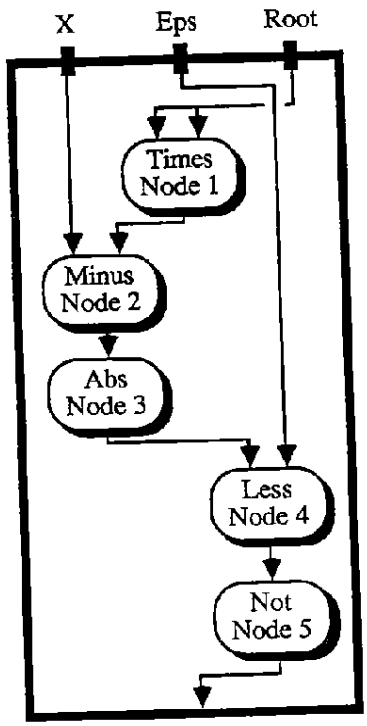
```
N 3   Abs
E     2 1   3 1   6
N 4   Less
E     3 1   4 1   6
E     0 2   4 2   6      %na=Eps
N 5   Not
E     4 1   5 1   1
E     5 1   0 1   1

G                 0      Returns (subgraph 3)
N 1   FinalValue
E     0 3   1 1   9      %na=Root
E     1 1   0 1   6

} 1   LoopA 4   0 2 1 3
E     0 1   1 1   6      %na=X
E     0 2   1 2   6      %na=Eps
E     1 1   0 1   6
```



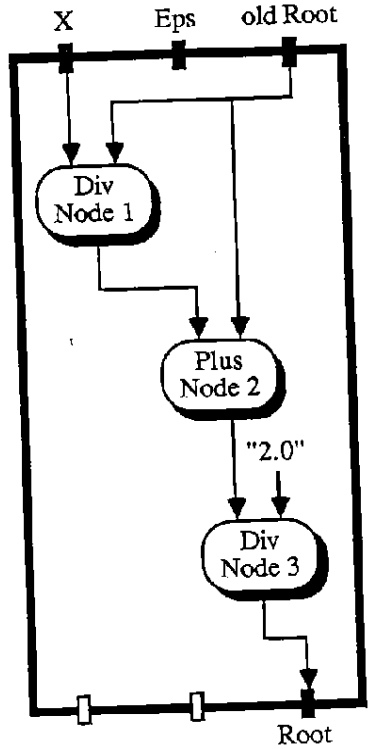
Graph of Function Sqrt



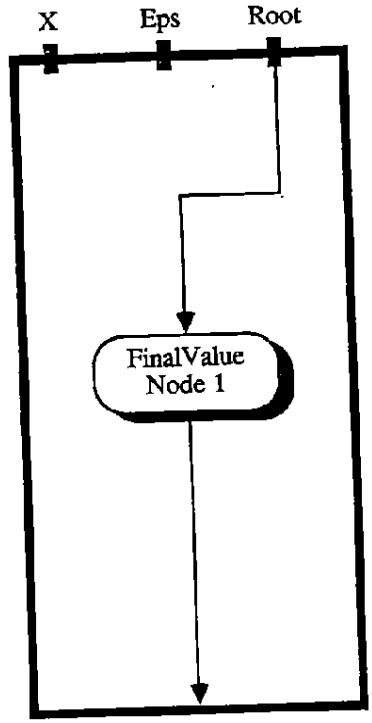
LoopA subgraph 0. Initialization



LoopA subgraph 1. Test



LoopA subgraph 2. Body



LoopA subgraph 3. Returns

Figure 10. Example of LoopA

5.2.5. LoopB

LOOPB
(value)+ → (value)+
4 subgraphs (Initialization, Test, Body, Returns)

LoopB is one of two iteration constructs (LoopA is the other). It repeatedly applies the loop body to a set of loop values, stopping and returning results when the test subgraph returns false. LoopB performs the termination test immediately after giving initial values to the loop values.

It has four subgraphs: initialization, test, body and returns. The association list's count field will always contain four, and the four integers that follow will give the number of the subgraph corresponding to Initialization, Test, Body and Returns, in that order.

If the test should return an error value, the objects returned by the returns subgraph must be in error also.

Implicit dependencies

(1) All inputs to the LoopB node are available to each subgraph on the class K ports.

(2) The loop values from the output of initialization are connected to the input of the test, body and returns subgraphs on the class L ports.

(3) The loop values from the output of the body are connected to the inputs of the test, body and returns subgraphs on the class L ports.

(4) The results of the returns subgraph are connected to the output ports of the LoopB node on the class R ports.

(5) When the result of the test subgraph is false, the LoopB makes its results available at its output ports.

Class	Usage	Port Range
K	values imported into the LoopB node	$1 .. n_K$
L	loop values	$n_K+1 .. n_K+n_L$
B	boolean value (terminate when false)	1 .. 1
R	results of the LoopA node	$1 .. n_R$

Table 16. Port Assignments for LoopB

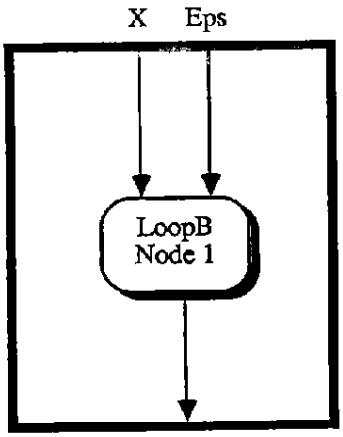
	Imports	Results
LoopB	K	R
Init	K	L
Test	K,L	B
Body	K,L	K,L
Returns	K,L	R

Table 17. Port Usage for LoopB

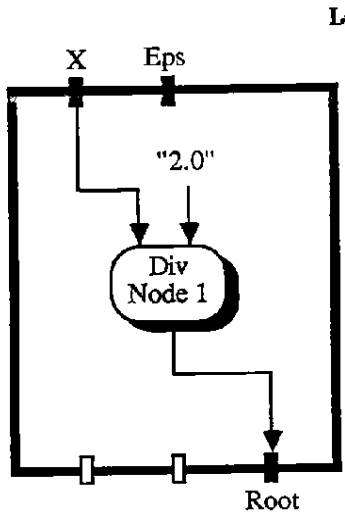
An Example:

```
T 1    Basic       0     %na=Boolean
T 2    Basic       1     %na=Character
T 3    Basic       2     %na=Double
T 4    Basic       3     %na=Integer
T 5    Basic       4     %na=Null
T 6    Basic       5     %na=Real
T 7    Multiple    1
T 8    Multiple    4
T 9    Multiple    6

C function Sqrt( X, Eps : real returns real )
C  for initial
C    Root := X/2.0
C  while abs(X - (Root*Root) ) >= Eps
C  repeat
C    Root := ( X /old Root + old Root ) / 2.0
C  returns value of Root
C  end for
C end function

T 10   Tuple       6     0
T 11   Tuple       6     1
T 12   Function    11    10

X                  12    "SQRT"
{ n_K=2 (X, Eps) n_L=1 (Root)

G                  0     Init (subgraph 0)
N 1    Div
E      0 1   1 1   6     %na=X
L            1 2   6     "2.0"
E      1 1   0 3   6     %na=Root

G                  0     Test (subgraph 1)
N 1    Times
E      0 3   1 1   6     %na=Root
E      0 3   1 2   6     %na=Root
N 2    Minus
E      0 1   2 1   6     %na=X
E      1 1   2 2   6
N 3    Abs
E      2 1   3 1   6
N 4    Less
E      3 1   4 1   6
E      0 2   4 2   6     %na=Eps
N 5    Not
E      4 1   5 1   1
E      5 1   0 1   1

G                  0     Body (subgraph 2)
N 1    Div
E      0 1   1 1   6     %na=X
E      0 3   1 2   6     %na=Root
```

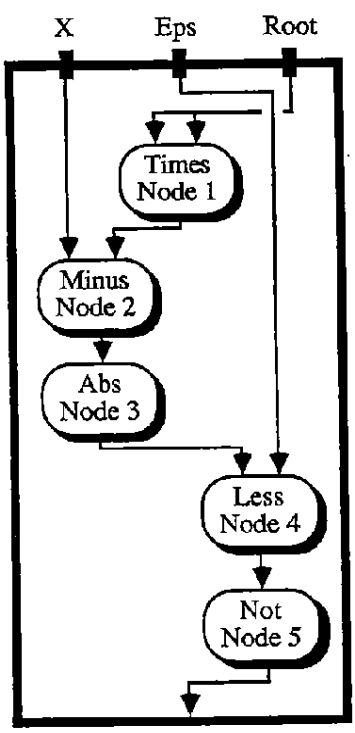
```
N 2   Plus
E     1 1   2 1   6
E     0 3   2 2   6      %na=Root
N 3   Div
E     2 1   3 1   6
L           3 2   6      "2.0"
E     3 1   0 3   6      %na=Root

G                 0      Returns (subgraph 3)
N 1   FinalValue
E     0 3   1 1   9      %na=Root
E     1 1   0 1   6

} 1   LoopB 4   0 1 2 3
E     0 1   1 1   6      %na=X
E     0 2   1 2   6      %na=Eps
E     1 1   0 1   6
```



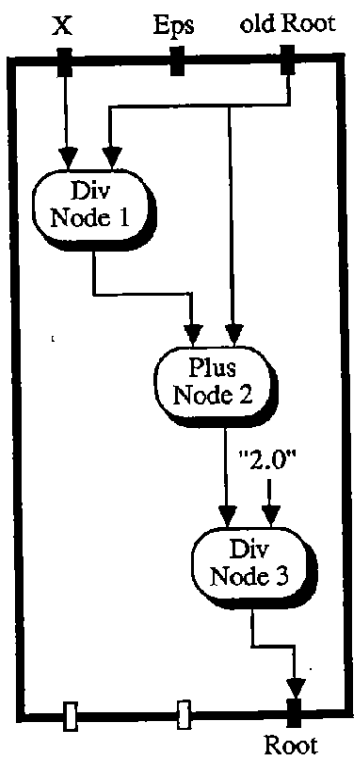
Graph of Function Sqrt



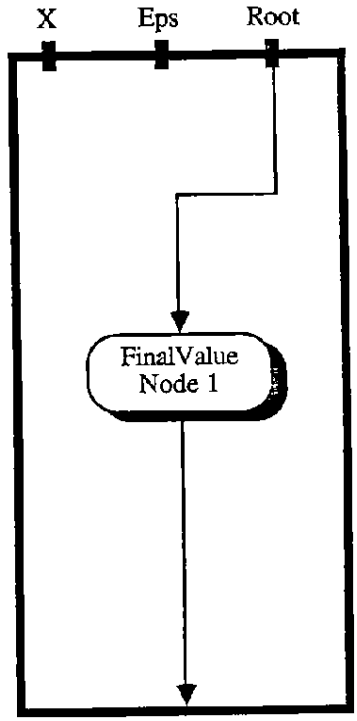
LoopB subgraph 0. Initialization



LoopB subgraph 1. Test



LoopB subgraph 2. Body



LoopB subgraph 3. Returns

Figure 11. Example of LoopB

6. References

[Ackermann 1979]
Ackermann, W. B. and J. B. Dennis, "VAL — A Value-Oriented Algorithmic Language: Preliminary Reference Manual", Massachusetts Institute of Technology, Laboratory for Computer Science, Technical Report 218, Cambridge, Massachusetts, June 1979.

[McGraw 1985]
McGraw, J. et al, "SISAL: Streams and Iteration in a Single-Assignment Language, Language Reference Manual", University of California, Lawrence Livermore National Laboratory, Report M-146, Rev. 1, Livermore, California, March 1985.

[Skedzielewski 1985]
Skedzielewski, S. K and R. K. Yates, "Fibre: An External Format for SISAL and IF1 Data Objects", University of California, Lawrence Livermore National Laboratory, Report M-154, Livermore, California, January 1985.

Appendix A

Lexemes of IF1

PosInteger	A decimal numeral representing an integer greater than zero
Integer	PosInteger + {'0'}
Literal	A literal constant as defined in Fibre (see Appendix B)
C, E, G, I, L, N, T, X, {, }, "	The ASCII characters themselves
Comment	A (possibly empty) sequence of ASCII characters except Newline
Newline	Whatever the operating system uses to separate lines
ErrorValue	error, undef, miss_elt, pos_overflow, etc. (the error values of the language, unadorned with type info)

BNF of IF1

Two meta-notations are present in the BNF. A list of one or more occurrences of a symbol is represented by " symbol... ", optional inclusions of symbols is written " [a b c] ", and zero or more occurrences of the proceeding would be written [a b c]... The number assigned to each kind of node is given in parentheses following the node's name. Similarly, numbers will be assigned to TypeTableEntries and BasicTypes. Terminal symbols of the grammar are denoted by **boldface** symbols.

Spaces or tabs separate lexemes in the IF1 language. They can also appear at the beginning or end of the line.

An IF1 file has the following form:

File ::= [Line **Comment Newline**...]...

Line ::=

C				
E	Source	Destination	TypeReference	
G			TypeReference	
G			TypeReference	**"Literal"**
I			TypeReference	**"Literal"**
L		Destination	TypeReference	**"Literal"**
L		Destination	TypeReference	**ErrorValue**
N	Label	Node		
T	Label	TypeTableEntry		
X			TypeReference	**"Literal"**
{				
}	Label	Node	Count	AssocList

```
Label           ::=  PosInteger

Source          ::=  SourceNode       SourcePort
Destination     ::=  DestinationNode  DestinationPort
SourceNode      ::=  Integer
SourcePort      ::=  PosInteger
DestinationNode ::=  Integer
DestinationPort ::=  PosInteger
TypeReference   ::=  Integer
Count           ::=  Integer
AssocList       ::=  Integer...

Node  ::=  Forall (0)
       |   Select (1)
       |   TagCase (2)
       |   LoopA (3)
       |   LoopB (4)

       |   AAddH (100)
       |   AAddL (101)
       |   AExtract (102)
       |   ABuild (103)
       |   ACatenate (104)
       |   AElement (105)
       |   AFill (106)
       |   AGather (107)
       |   AIsEmpty (108)
       |   ALimH (109)
       |   ALimL (110)
       |   ARemH (111)
       |   ARemL (112)
       |   AReplace (113)
       |   AScatter (114)
       |   ASetL (115)
       |   ASize (116)
       |   Abs (117)
       |   BindArguments (118)
       |   Bool (119)
       |   Call (120)
       |   Char (121)
       |   Div (122)
       |   Double (123)
       |   Equal (124)
       |   Exp (125)
       |   FirstValue (126)
       |   FinalValue (127)
       |   Floor (128)
       |   Int (129)
       |   IsError (130)
       |   Less (131)
       |   LessEqual (132)
       |   Max (133)
       |   Min (134)
       |   Minus (135)
```

```
                | Mod (136)
                | Neg (137)
                | NoOp (138)
                | Not (139)
                | NotEqual (140)
                | Plus (141)
                | RangeGenerate (142)
                | RBuild (143)
                | RElements (144)
                | RReplace (145)
                | RedLeft (146)
                | RedRight (147)
                | RedTree (148)
                | Reduce (149)
                | RestValues (150)
                | Single (151)
                | Times (152)
                | Trunc (153)

TypeTableEntry  ::=  Array (0)      TypeReference
                     | Basic (1)    BasicType
                     | Field (2)    TypeReference   TypeReference
                     | Function (3) TypeReference   TypeReference
                     | Multiple (4) TypeReference
                     | Record (5)   TypeReference
                     | Stream (6)   TypeReference
                     | Tag (7)      TypeReference   TypeReference
                     | Tuple (8)    TypeReference   TypeReference
                     | Union (9)    TypeReference

BasicType  ::=  Boolean (0)
                | Character (1)
                | Double (2)
                | Integer (3)
                | Null (4)
                | Real (5)
```

Appendix B

Literal Values in IF1

Literals in IF1 use the Fibre format for data objects [Skedzielewski, 1985].

```
Literal                ::=  Integer
                          |  Single
                          |  Double
                          |  Character
                          |  Boolean
                          |  Null
                          |  Array
                          |  Stream
                          |  Record
                          |  Union
PrintableCharacter     ::=  ' ' through '~'
PrintableNonBackslash  ::=  PrintableCharacter less '\'
OctalDigit             ::=  0|1|2|3|4|5|6|7
Digit                  ::=  OctalDigit | 8|9
Minus                  ::=  -
Integer                ::=  [ Minus ] Digit...
Single                 ::=  [ Minus ] Digit... . [ Digit ]... [ Power ]
                          |  [ Minus ] . Digit... [ Power ]
                          |  [ Minus ] Digit... Power
Power                  ::=  Exp [ Minus ] Digit...
Exp                    ::=  e|E
Double                 ::=  [ Minus ] Digit... . [ Digit ]... DPower
                          |  [ Minus ] . Digit... DPower
                          |  [ Minus ] Digit... DPower
DPower                 ::=  Dexp [ Minus ] Digit...
Dexp                   ::=  d|D
Char                   ::=  'PrintableNonBackslash'
                          |  '\ PrintableCharacter'
                          |  '\ OctalDigit...'
Boolean                ::=  T|F
Array                  ::=  [ Integer : [ ValueList ] ]
                          |  "PrintableCharacter..."
ValueList              ::=  Value... [ MoreElements ]
MoreElements           ::=  ; Integer : ValueList
Stream                 ::=  { Integer : [ ValueList ] }
Record                 ::=  < Value... >
Union                  ::=  ( Integer : Value )
Null                   ::=  NIL
```

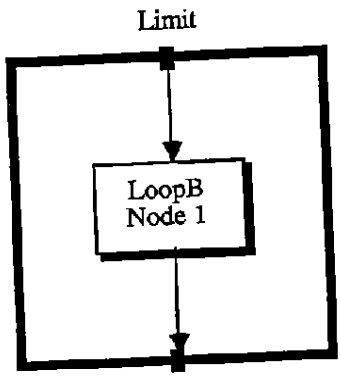
Appendix C

An IF1 Example

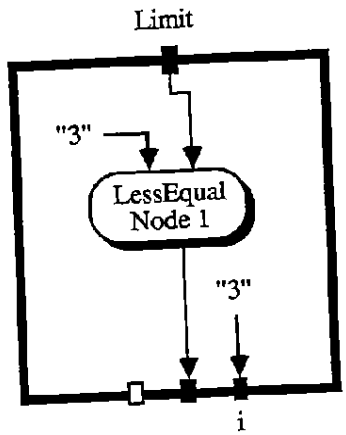
This section contains a translation of the "Sieve of Eratosthenes" example in Appendix D of the SISAL reference manual.

```
C "standard" types
T 1   Basic       0     %na=Boolean
T 2   Basic       1     %na=Character
T 3   Basic       2     %na=Double
T 4   Basic       3     %na=Integer
T 5   Basic       4     %na=Null
T 6   Basic       5     %na=Real
T 9   Multiple    1
T 10  Multiple    4

C type Si = stream [integer]
T 11  Stream            4

C global Sqrt ( Q:real returns real )
T 12  Tuple       6     0
T 13  Function    12    12
I                 13    "Sqrt"

C function OddIntegers ( Limit : integer returns Si )
C  for initial
C    i := 3
C  while i <= Limit
C  repeat
C     i := old i + 2
C  returns stream of i when ( i <= Limit )
C  end for
C end function

T 14  Tuple       4     0
T 15  Tuple       11    0
T 16  Function    14    15


G                 16    "OddIntegers"
E     1 1   0 1   11
{ n_K=1 (Limit) n_L=1 (i)

G                 0     Init (subgraph 0)
E     1 1   2 1
L           0 3   4     "3"
N 1   LessEqual
L           1 1   4     "3"
E     0 1   1 2   4     %na=Limit
```

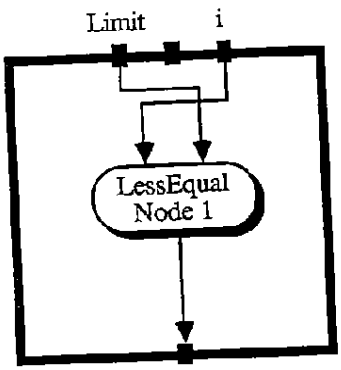
```
G                       0     Test (subgraph 1)
E     1 1   0 1         1
N 1   LessEqual
E     0 3   1 1         4     %na=i
E     0 1   1 2         4     %na=Limit

G                       0     Body (subgraph 2)
E     2 1   0 2         1
E     1 1   0 3         4     %na=i
N 1   Plus
E     0 3   1 1         4     %na=i
L           1 2         4     "2"
N 2   LessEqual
E     1 1   2 1         4     %na=i
E     0 1   2 2         4     %na=Limit

G                       0     Returns (subgraph 3)
E     1 1   0 1         11
N 1   AGather
L           1 1         4     "1"
E     0 3   1 2         10    %na=i
E     0 2   1 3         9

} 1   LoopB 4  0 1 2 3
E     0 1   1 1         4     %na=Limit
```



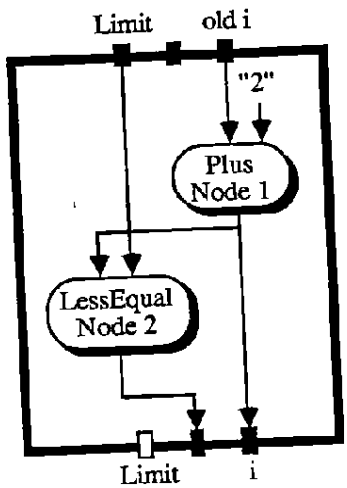
Graph of function OddIntegers



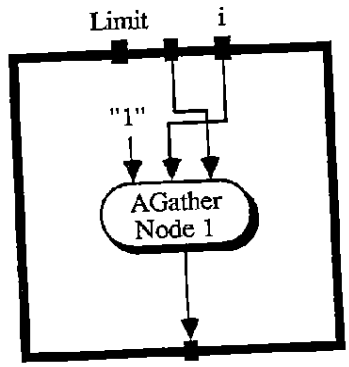
LoopB subgraph 0. Initialization



LoopB subgraph 1. Test



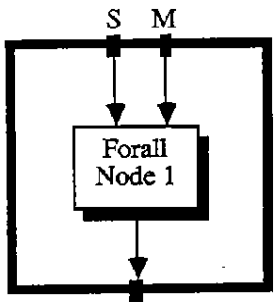
LoopB subgraph 2. Body



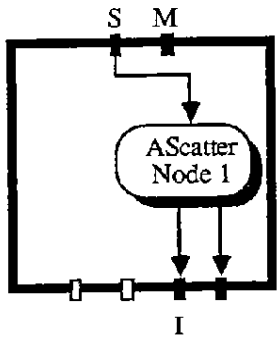
LoopB subgraph 3. Returns

Figure 12. Function OddIntegers

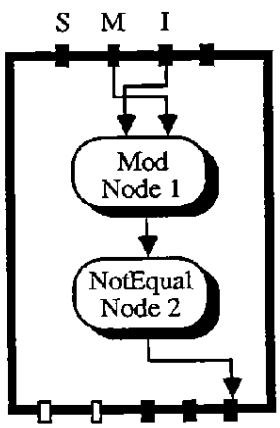
```
C function Filter ( S : Si; M : integer returns Si )
C for I in S
C   Multiple : boolean := mod (I,M) = 0
C returns stream of I unless Multiple
C end for
C end function

T 17  Tuple          11     14
T 18  Function       17     15

G                    18     "Filter"
E     1 1    0 1     11
{ n_K=2 (S,M) n_L=2 (Multiple,noname) n_T=0

G                    0      Generator (subgraph 0)
E     1 1    0 3     10     %na=I
E     1 2    0 4     10     %na=noname
N 1   AScatter
E     0 1    1 1     11     %na=S

G                    0      Body (subgraph 1)
E     3 1    0 5     1
E     2 1    0 6     1      %na=Multiple
N 1   Mod
E     0 3    1 1     4      %na=I
E     0 2    1 2     4      %na=M
N 2   NotEqual
E     1 1    2 1     4
L              2 2     4     "0"
N 3   Not
E     2 1    3 1     1      %na=Multiple

G                    0      Returns (subgraph 2)
E     1 1    0 1     11
N 1   AGather
L              1 1     4    "1"
E     0 3    1 2     10     %na=I
E     0 5    1 3     9

} 1   Forall 3   0 1 2
E     0 1    1 1     11     %na=S
E     0 2    1 2     4      %na=M
```



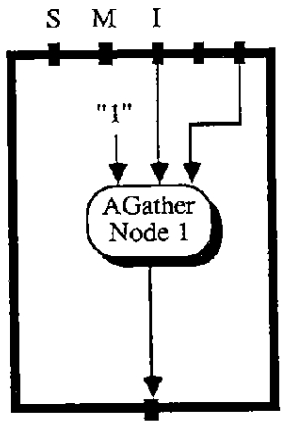
Graph of function Filter



Forall subgraph 0. Generator



Forall subgraph 1. Body



Forall subgraph 2. Returns

Figure 13. Function Filter

```
C function Sieve ( Limit : integer returns Si )
C % returns the stream of prime numbers up to Limit
C  let
C    Maxt := integer ( Sqrt ( real ( Limit ) ) );
C    Primes := for initial
C              S := OddIntegers ( Limit );
C              T := 1
C            while ~stream_empty ( S )
C            repeat
C              T := stream_first ( old S );
C              S := if T <= Maxt
C                   then Filter ( stream_rest ( old S ), T )
C                   else stream_rest ( old S )
C                  end if
C            returns stream of T
C            end for
C  in
C    stream_rest( Primes )
C  end let
C end function
```

```
X                       16      "Sieve"
E      5 1    0 1       11
N 1    Single
E      0 1    1 1       4       %na=Limit
N 2    Call
L              2 1       13      "Sqrt"
E      1 1    2 2       6
N 3    Floor
E      2 1    3 1       6
{ n_K=2 (Limit,Maxt) n_L=2 (S,T) n_T=0

G                       0       Init (subgraph 0)
E      1 1    0 3       11      %na=S
L              0 4       4       "1" %na=T
N 1    Call
L              1 1       16      "OddIntegers"
E      0 1    1 2       4       %na=Limit

G                       0       Test (subgraph 1)
E      2 1    0 1       1
N 1    AIsEmpty
E      0 3    1 1       11      %na=S
N 2    Not
E      1 1    2 1       1

G                       0       Body (subgraph 2)
E      2 1    0 3       11      %na=S
E      1 1    0 4       4       %na=T
N 1    AElement
E      0 3    1 1       11      %na=S
L              1 2       4       "1"
{ n_K=3 (Maxt,old S,T)
```

```
G                       0       Predicate (subgraph 0)
E     2 1   0 1   4
N 1   LessEqual
E     0 3   1 1   4     %na=T
E     0 1   1 2   4     %na=Maxt
N 2   Int
E     1 1   2 1   1

G                       0       False Branch (subgraph 1)
E     1 1   0 1   11
N 1   ARemL
E     0 2   1 1   11    %na=old-S

G                       0       True Branch (subgraph 2)
E     2 1   0 1   11
N 1   ARemL
E     0 2   1 1   11    %na=old-S
N 2   Call
L           2 1   18    "Filter"
E     1 1   2 2   11
E     0 3   2 3   4     %na=T

} 2   Select 3   0 1 2
E     0 2   2 1   4     %na=Maxt
E     0 3   2 2   11    %na=old-S
E     1 1   2 3   4     %na=T

G                       0       Returns (subgraph 3)
E     1 1   0 1   11
N 1   AGather
L           1 1   4     "1"
E     0 4   1 2   10    %na=T

} 4   LoopB 4   0 1 2 3
E     0 1   4 1   4     %na=Limit
E     3 1   4 2   4     %na=Maxt
N 5   ARemL
E     4 1   5 1   11    %na=Primes
```

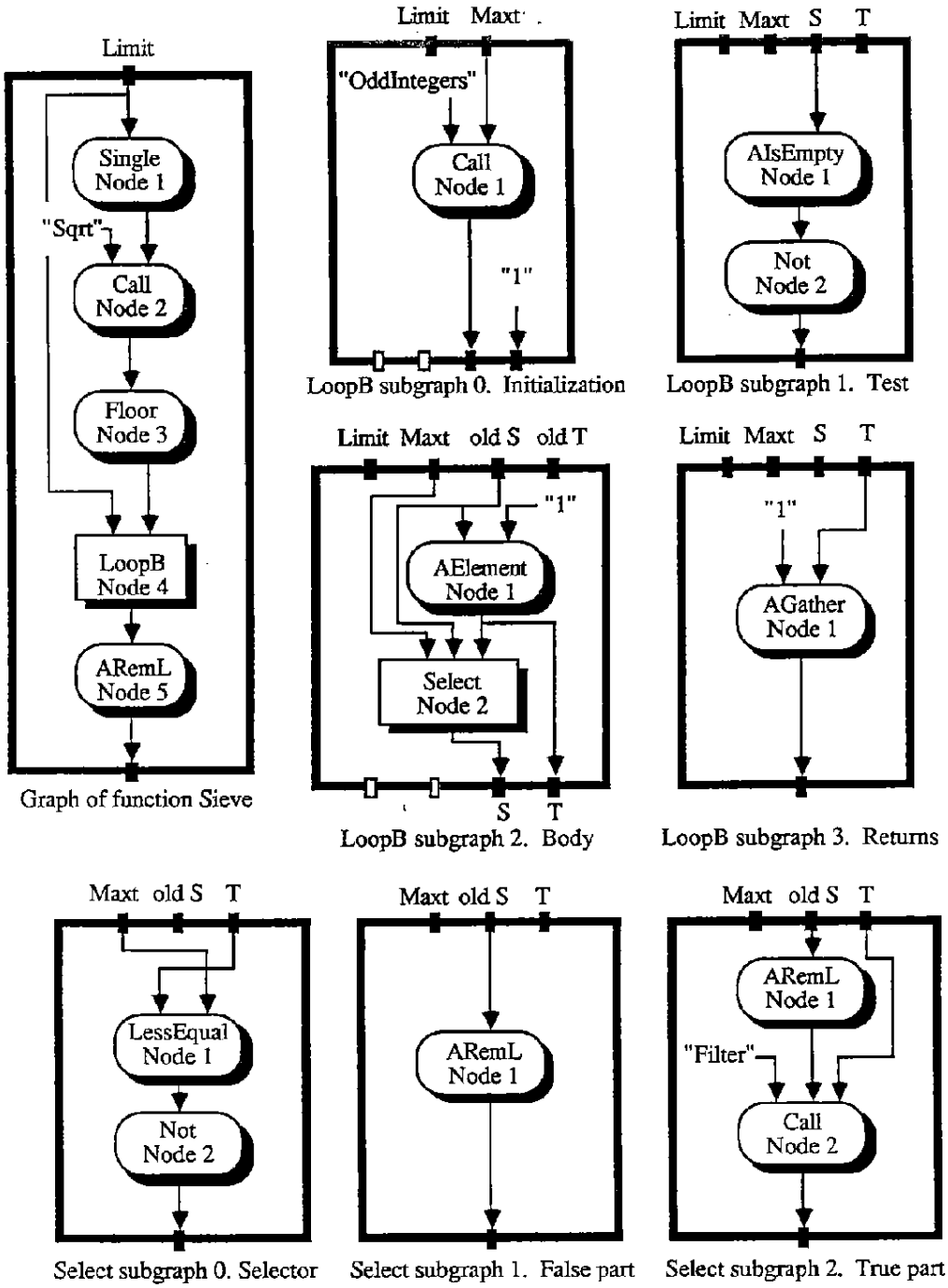


Figure 14. Function Sieve

Appendix D

Hints for the SISAL Implementor

Multi-dimensional array operations

All array operations in IF1 are done on only one dimension at a time. Thus, n AElement nodes are needed to access an element of an n-dimensional array, and quite complicated graph structures are needed for the array replace operation on multi-dimensional arrays. An example of the latter is given in the section on array manipulation nodes.

if-then-elseif-else

Since the predicate of an if-then-else without conjunctions/disjunctions is always computed (in contrast to the expressions in its alternatives) it is advantageous to pull such predicates out of the selector subgraph of a Select node. By doing so, some code improvers (e.g. common subexpression eliminators) will do a better job.

The use of **elseif** is represented in IF1 by nesting another Select node in the "false" subgraph of the select. Naturally, the nesting can become quite deep if several elseifs are present in the expression.

Forall Expressions

Forall generator subgraphs should contain only AScatter and RangeGen nodes. Dot products will place one AScatter/RangeGen in the generator for each value in the dotted list, otherwise just one node generates the loop values.

Cross products are represented in IF1 by nested Forall nodes.

The returns subgraph should contain only nodes that take *multiple* types as input. Expressions in SISAL **returns** clauses may cause code to appear in the body subgraph or outside the Forall node, rather than appearing in the returns subgraph (since those expressions may need operators whose inputs are not *multiples*). For example, if A and B are values generated in the Forall body,

```
returns value of A*B
```

should cause the Forall to produce *two* results (the final values of A and B), and multiply them following the Forall node.

```
returns array of A*B
```

would cause the Forall body subgraph to produce one output (A*B), which is sent into an AGather in the returns subgraph. Note that this code motion is necessary for all returns clauses *except* "value of" clauses (array of, stream of, reductions).

For Initial Expressions

The "for initial" expression generates a LoopA or LoopB node; these nodes cannot contain RangeGen or AScatter nodes.

The returns subgraph collects its first values from the initialization subgraph, and later values from all iterations of the body subgraph.

The code motion for expressions in returns parts is similar to that needed in forall expressions, except that when code is moved into the body subgraph, it must also be moved into the initialization subgraph.

Nodes in Returns Subgraphs

Results of **for** and **forall** expressions can only be produced by Reduce, AGather, or FinalValue. Special care must be taken to implement SISAL semantics correctly, watching out for zero-trip and broken loops. AGather can be initialized to an empty array or stream, and elements added if and when they appear.

Reduce can be initialized to the unit value of the reduction function, except that "least", "greatest", and "catenate" require special handling. Generally, zero-element reductions and gathers yield the unit value.

Reductions of **least** and **greatest** must return *error* if no elements are reduced. Reductions of **catenate** yield an empty array if no values are reduced, but care must be taken with the first array to ensure that correct lower bound and prefix size are created for the result array.

Note well that these situations can occur in LoopA and LoopB nodes, as well as in Forall, when a filter is present on the Reduce node.

Of course, broken loops (error input to generators, error values for tests) always mark the results as *error* and terminate.

Setting the Lower Bound of a Gathered Array

The AGather node requires a lower bound for its result. You can either use the FirstValue node on the multiple indices (produced by the generator of interest), or you can set the lower bound to one in the AGather, and then reset the lower bound in a separate operation outside the compound node. The latter is slightly preferred by the LLNL software.

old Modifier in Returns Parts

All of the clauses that use the old modifier must filter any *multiple* values that they use with the AllButLast node.

Name field for 'old' values

The name pragma for "old LoopName" should be written "%na=old-LoopName".

Error Semantics

The error semantics of operations that produce objects with basic types are simple; return error[basic type]. However, the semantics of error values for compound types are more complicated and are described in great detail in Appendix F of the SISAL language reference manual [McGraw 1985].

Boolean Expressions

Be aware that Boolean expressions in SISAL are defined to be lazily evaluated (see the bottom of page 5-1 of the reference manual). Lazy evaluation requires that Boolean operations on error values be handled differently from other operations on error values. For example, 'false & error[Boolean]' yields false, whereas '0 + error[integer]' yields error[integer].