lawrence livermore
national laboratory

M-146
Rev. 1

Sisal:

Streams and Iteration in a Single Assignment Language

Language Reference Manual Version 1.2

March 1, 1985

Work performed under the auspices of the U. S. Department of Energy by the UCLLNL under contract No. W-7405-ENG-48.

university of california • davis

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

SISAL

Streams and Iteration in a Single Assignment Language

Language Reference Manual
Version 1.2

March 1, 1985

Authors:

James McGraw, LLNL
Stephen Skedzielewski, LLNL

Stephen Allan, CSU
Rod Oldehoeft, CSU

John Glauert, UEA

Chris Kirkham, UM

Bill Noyce, DEC
Robert Thomas, DEC

Table of Contents

1. INTRODUCTION

Many multi-processor systems are currently under study by various groups around the world. The understanding and exploitation of parallelism in these systems is a primary goal of these studies. To facilitate the use and comparison of these systems, we proposed to :

- Define a common high-level language :
  The primary candidate was a single-assignment, applicative, dataflow language as defined in [1, 2, 3]. In spite of remarkable diversity in hardware structures, all proposed systems could benefit from the functional semantics of such a language, and from other characteristics such as implicit parallelism, freedom from side effects, locality of effects, etc.

- Produce a compiler having :
  - a single "Front-end" (language specific) parser and
  - several "Back-end" (machine specific) code generators

- Define an Intermediate Format (IF) [4], to serve as the
  - interface between parser and code generators
  - interface between this system and other language systems

- Define an External Format (Fibre) [5], to provide :
  - a simple interface for specifying inputs and outputs to SISAL and IF graphs

- To develop and share a pool of benchmark programs

The proposed system is shown in Figure 1.

A cooperative effort of the Colorado State University, DEC, Lawrence Livermore National Laboratory, and University of Manchester has resulted in the programming language SISAL which is described in this manual.

SISAL (Streams and Iteration in a Single-Assignment Language) is a functional data-flow language intended for use on a variety of sequential, vector, multi-, and data-flow processors. Our primary goal is to produce a compiler hosted on the VAX and targeted to both the VAX and the other machines. Secondary goals are:

1) promote wide use of the language in parallel processing research centers, i.e., produce a de facto standard,
2) allow study of architectural trade-offs inherent in machine design
3) provide a vehicle for developing and sharing a pool of benchmarks
4) allow study of the benefits or lack thereof of a functional programming style.

SISAL is designed to express algorithms for execution on computers capable of highly concurrent operation. More specifically, the application area to be supported is numerical computation which strains the limits of high performance machines, and the primary targets for translation of SISAL programs are dataflow data-driven machines .

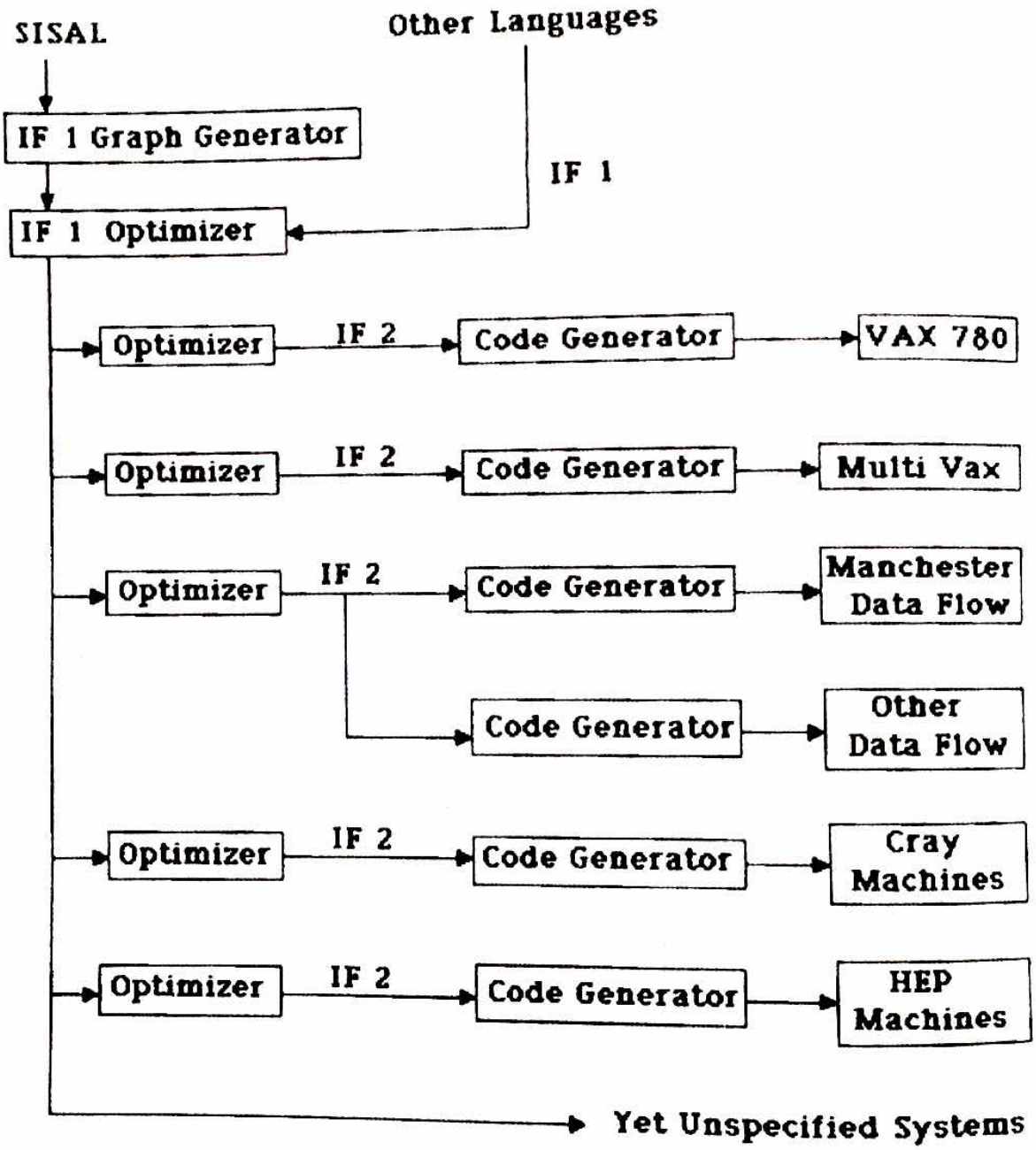
SISAL
Other Languages
IF 1 Graph Generator
IF 1
IF 1 Optimizer
Optimizer
IF 2
Code Generator
VAX 780
Optimizer
IF 2
Code Generator
Multi Vax
Optimizer
IF 2
Code Generator
Manchester Data Flow
Code Generator
Other Data Flow
Optimizer
IF 2
Code Generator
Cray Machines
Optimizer
IF 2
Code Generator
HEP Machines
Yet Unspecified Systems

Figure 1

Nevertheless, it has been our intention that the language not have idiosyncrasies reflecting the particular nature of the application area or target machine. It should be reasonable for SISAL to evolve into a general purpose language appropriate for writing programs to run on future general parallel computers.

We have undertaken the design of a new language because existing languages for numerical computation have a serious deficiency: they reflect the storage structure of the von Neumann concept of computer organization in that each language has some method of effecting a change in state of the memory that cannot be modeled as a local effect. FORTRAN, still the most popular language for large scale numerical work, is particularly blatant in this respect since it was conceived as a high level notation for programs to be run on a machine of classical design (the IBM 704).

Languages allowing the specification of global state changes lead to programs that are very difficult (or impossible) to analyze for parts that may be executed concurrently. It is impossible in general to trace the flow of data with less than a complete analysis of the entire program. Only with such analysis is it possible to find and eliminate inessential constraints on the sequencing of program parts.

In contrast, the language SISAL is entirely free of side effects: each module or well formed portion of a SISAL program corresponds to a mathematical function and the entire effect of putting two parts together is to compose the corresponding functions. Such a language is functional or applicative. Although designs for applicative languages have been discussed many times in the literature, there have been few attempts to construct a complete and practical definition. This is due to the difficulty of incorporating file updates and input/output operations within the applicative framework, and the question of efficiency of implementation. The efficiency issue is countered in SISAL by our goal of highly parallel execution, which is supported by applicative languages, and our aim to develop computer architectures specifically for efficient execution of programs expressed in functional languages.

The file update and input/output issues are addressed through the introduction of streams of values as a principal means for communicating between program modules. Modules that produce streams as output or accept streams as input can be used for input/output processes. Furthermore, the implementation of transactions on a data base may be viewed as the processing of a stream of commands by a data base "secretary" or "guardian" module that holds the data base as internal data. If it is desired to realize more concurrency in processing transactions, the data base may be divided into parts, each with its own secretary module.

1.1. Acknowledgements

In developing the definition of SISAL, we started from a language design which is well documented, and closest to meeting our goals. Such a language is VAL, developed at MIT by the Computation Structures Group of the Laboratory for Computer Science under Jack Dennis [1]. We thank him for many of the fundamental ideas of SISAL. We are also most grateful for his permission to use the VAL reference manual text as the starting point for the definition of this manual. This short-cut has allowed us to focus the additions and changes we felt were needed.

The first draft of this report was completed during the SISAL workshop at DEC, Hudson, Mass. Feb.17-18, 1983. Several workshop participants have influenced the development of SISAL through their criticism. These are Arvind, Jack Dennis, Joe Fasel, Maya Gokhale, Alan Hayes, Vinod Kathail, and Shane Robison.

We are thankful to George Michael of the Lawrence Livermore National Laboratory and Bill Keating, Leslie Klein, Steve Teicher, and Alain Hanover of Digital Equipment Corp. for their support.

Special thanks go to Robert Yates for his efforts in formatting and proofreading this document.

The changes introduced in Version 1.2 were the product of a meeting hosted by Allan and Oldehoeft at Colorado State University on Feb. 13-15, 1984. Participants there included Kim Yates, Michael Welcome, John Lang, and Bruce Bigler. (in addition to the authors). All changes introduced in this version of the manual will be identified by a vertical bar in the right-hand column of all lines that have been changed.

1.2. References

[1] W.B.Ackerman, J.B.Dennis, "VAL -- A value-oriented algorithmic language: Preliminary reference manual," Tech.report TR-218, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., June 1979.

[2] J.R.McGraw, "The VAL Language : Description and Analysis," ACM Trans. on Programming Languages and Systems, Jan.82, Vol.4, No.1, pp.44-82.

[3] J.R.McGraw, S.K.Skedzielewski ,"Streams and Iteration in VAL : Additions to a Data Flow Language," Proceedings- Int. Conf. on Distributed Computing Systems, Miami, Fl., March 1983. (IEEE Order No. CH 18028)

[4] S.K.Skedzielewski, J.Glauert, "IF1, an Intermediate Form for Applicative Languages", Lawrence Livermore National Laboratory Technical Report (in progress).

[5] S.K.Skedzielewski, R.Kim Yates, "Fibre: An External Format for SISAL and IF1 Data Objects", Lawrence Livermore National Laboratory Technical Report (in progress).

1.3. Addresses of Authors or Current Contacts

Name / Address	Telephone	Arpanet Address
James McGraw Stephen Skedzielewski Lawrence Livermore National Laboratory Box 808, L-306 Livermore, California 94550	(415) 422-0541 423-1516	McGraw at lll-crg Skedz at lll-crg
Stephen Allan Dale Grit Rod Oldehoeft Colorado State University Dept. of Computer Science Fort Collins, Colorado 80523	(303) 491-5373 491-7033 491-7017	Allan.ColoState at csnet-relay Grit.ColoState at csnet-relay RRO.ColoState at csnet-relay
John Glauert School of Information Sciences University of East Anglia Norwich NR4 7TJ England	(44) 603-56161 ext. 204	Glauert%eapa at ucl-cs
Chris Kirkham University of Manchester Computer Center Oxford Road Manchester, M13 9PL England	(44) 61-273-1993	id.Gurd at mit-xx
Bill Noyce Digital Equipment Corp. ZK02-3/N30 110 Spitbrook Road Nashua, NH 03062	(603) 881-2050	Noyce%eludom.dec at decwrl

2. LANGUAGE SUMMARY

A program in SISAL is a collection of separately translated parts called compilation units. Each compilation unit defines an arbitrary number of functions and the nature of the interface that compilation unit is to have with other compilation units. The interface identifies the declared functions that are to be visible (**defined**) outside of the current unit and the interface also describes the functions used by this unit that may not be defined there (**global**). All functions in SISAL must be defined before they can be used. Mutually recursive functions are permitted, in which case one of the functions must be first described in a **forward** definition.

The SISAL language is applicative, that is, value-oriented. In contrast to many other languages, there are no "objects" thought of as residing in memory and being updated as the computation progresses. Even arrays and records are treated in SISAL as mathematical values.

A function computes one or more data values as a function of one or more argument values. Except for invocations of other functions, a function invocation has access only to its arguments; there are no side effects. Further, a function retains no state information from one invocation to another; each function invocation is strictly independent. Hence values returned by a function depend only on the argument values presented to it -- a SISAL function implements a true function in the mathematical sense.

The data types of SISAL include the basic scalar types boolean, integer, real, double real, null and character. Data structure values can be record values, array values, union values, or stream values. Records have a fixed format; each field has a specified type. An array has an integer index set and its components are of arbitrary but uniform type. A stream is an ordered sequence of values of arbitrary but uniform type. Components of a stream are accessible only through a set of built-in functions (e.g., first and rest). Union types may be formed in which tags allow discrimination among a specified set of constituent types. The value of any object of a union type consists of a tag name and a value whose type is one of the constituent types. Data structures of arbitrary depth may be specified using nested array, stream, record, and union types.

Each data type has its associated set of operations and predicates. Array, stream, union, and record types are treated as mathematical sets of values -- just as the basic scalar types. The operations for arrays, streams, unions, and records are chosen to support identification of concurrency for execution on a highly parallel processor.

SISAL handles exceptional conditions by producing special error values that serve two purposes. First, an error value indicates that some kind of computational difficulty prevented the correct production of a normal answer. Second, an error value may preserve portions of structured objects that are known to be correct (i.e., not affected by the computational difficulty). The value **error** is a proper element of every SISAL type. This value will be produced in the event of arithmetic or control flow errors. Arithmetic errors that could produce this value include: overflow, underflow, divide by zero, array subscript out of bounds, etc. . Control flow errors include: conditional expressions whose test clause produces an error, iterations whose termination test produces an error, etc. . SISAL also defines rules for propagating these error values through any expression. In general, an error value cannot be canceled out of a calculation without

explicit program testing (e.g., **error** * 0 yields **error**). However, every effort is made to preserve partial results in the face of erroneous calculations. For example, if an iteration is to produce an array and the termination test produces an error, the resulting array has the property of being an **error**, but accesses to portions of the array that were correctly computed prior to the error will yield those values.

The design of SISAL permits type checking to be performed by the translator. The type of each argument or result value of a function is specified in the function definition's header. The type of each value name used in the body of a function is always directly inferable from the context in which it is used. The operations of SISAL are designed so that the types of the results can be determined if the types of the operands are known. Since the types of all atomic expressions are manifest, the types of all expressions can be determined. For purposes of type checking, SISAL uses structural type matching and does not perform any automatic type coercions.

Since SISAL is a side-effect free language, subexpressions may be evaluated in any order without effect on computed results. Thus the control structures of SISAL use a syntactic form -- an expression -- evaluation of which yields a tuple of values. Language constructs are provided for conditional expressions (**if/then/else**) and for iteration expressions (**for**). In addition, expression structures are provided for distributed computation of the components of a new array, stream, or of values to be combined by an operator. Variants of the **for** expression may be used to compute the component values of a new array or stream simultaneously, or to combine simultaneously computed values by an associative operation such as addition, multiplication, or maximum.

2.1. Notation

In the BNF presentation of the syntax, square brackets denote optional material. Ellipses (...) indicate that the preceding syntactic unit may be repeated one or more times. Comments on the syntax appear within parentheses and are not part of the grammar. Reserved words and special symbols that are part of the SISAL language (rather than part of the meta-notation) will appear in bold face type. The following examples illustrate some of the common combinations.

decldef-part	::=	decldef [; decldef] ...	(non-empty list of decldef's separated by semicolons)
array-ref	::=	primary [expression]	(the square brackets here must appear as part of the array reference in a SISAL program)

3. LEXICAL CONVENTIONS

Programs are written using the ASCII character set. No "control" characters other than tab and newline are used within the source. The program elements are:

operation and punctuation symbols
real and integer numbers
character and character string constants
reserved words
names
comments

The operation and punctuation symbols are the following:

```
+  -  *  /  |  &  ~
||  <  >  <=  >=  =
~=  :=  :  .  ;  ,
(  )  [  ]  '  "
```

An integer number is a non-empty sequence of decimal digits without a decimal point. The type of an integer number is assumed to be **integer**.

A real number is an integer number followed by one of:

1. a decimal point
2. an exponent field
3. a decimal point followed by an exponent field
4. a decimal point followed by an integer number
5. a decimal point followed by an integer number followed by an exponent field.

An exponent field is the letter 'E', 'e', 'D' or 'd', an optional sign, and one or more decimal digits. Real numbers without an exponent field, or which use the letter 'E' or 'e' in the exponent field are assumed to be of type **real**. Real numbers using the letter 'D' or 'd' in the exponent field are assumed to be of type **double_real**.

A character constant is a single character enclosed in single quotes. A character string constant is a sequence of zero or more characters enclosed in double quotes. Neither a character nor a character string constant may extend across a line boundary.

The type of a character constant is **character**. The type of a string constant is **array[character]**, where the low bound is one and the high bound is the number of characters in the string.

Control characters may be inserted into a character constant or string by the use of a notation taken from the C programming language. The backslash character, when immediately followed by a character in the set { n r t f b \ " ' }

denotes:

\n	newline	(implementation dependent)
\r	carriage return	ASCII CR
\t	horizontal tab	ASCII HT
\f	form feed	ASCII FF
\b	backspace	ASCII BS
\\	backslash	
\"	double quote	(when it is a part of the string)
\'	apostrophe	(for use in a character constant)

Control characters may be inserted into a character or character string constant by following a backslash with three octal digits. Thus \000 represents NUL, \007 is a bell, etc.

If the backslash is followed by something other than one of the above characters, the result is the character that follows the backslash (e.g., \s yields s).

Each of the above characters is exactly one character "in width".

A reserved word is a word that always has a special meaning. Reserved words may never be used in any context for other than their special meaning. Reserved words are printed in boldface when they appear in program examples and the syntax description of this report.

The reserved words are:

array	**at**	**boolean**	**catenate**	**character**
cross	**define**	**dot**	**double_real**	**else**
elseif	**end**	**error**	**false**	**for**
forward	**function**	**global**	**greatest**	**if**
in	**initial**	**integer**	**is**	**least**
left	**let**	**nil**	**null**	**of**
old	**otherwise**	**product**	**real**	**record**
repeat	**replace**	**returns**	**right**	**stream**
sum	**tag**	**tagcase**	**then**	**tree**
true	**type**	**union**	**unless**	**until**
value	**while**	**when**		

The following names are predefined functions in SISAL, but they are not reserved words:

abs	array_addh	array_addl	array_adjust
array_fill	array_limh	array_liml	array_prefixsize
array_remh	array_reml	array_setl	array_size
exp	floor	max	min
mod	stream_append	stream_empty	stream_first
stream_prefixsize	stream_rest	stream_size	trunc

A name is a sequence of letters, digits, and underscores, of which the first character must be a letter. A name may not be the same as a reserved word. A name may be used as a value name, a function name, a defined type name, a record field name, or a union tag name. These uses all have their own mechanisms for interpretation, and hence a name may be used without conflict for

several purposes. For example:

```
type Complex = record[ Re, Im : real ]
type Im = boolean
```

is a legal set of type definitions, even though the name Im appears twice. Context information is used to distinguish them.

Upper and lower case letters in names and reserved words are not distinguished. Names may be of any reasonable length. Characters after the thirty-first are, however, ignored.

The separating characters space, tab, and newline are equivalent (except in delimiting comments). They may appear anywhere a space may appear (hence they may not appear within a number or between the characters of a two character operation symbol such as >=). A separating character is required only between adjacent constants, names, or reserved words. For example, separating characters are required to distinguish the program construct "if p then 3 else 4 end if" from the name "ifpthen3else4endif".

A comment begins with a percent sign and continues to the end of the line. A comment is equivalent to a space, and may appear anywhere that a space may appear.

If the character immediately following the percent sign in a comment is a dollar sign the comment is considered to be a compiler directive, or pragma. See Appendix C for a description of compiler directives.

Examples of names and constants:

```
ABC3_Q                  % name
ABC3_q                  % the same name
34                      % integer number
0.3141593E1             % real number
0.3141593D1             % double real number
2.718282                % real number
5772157E-7              % real number
'%'                     % character constant
'\''                    % character constant
"abc\"\ndef"            % character string constant, length 8
"abc\"\nd\ef"           % the same character string constant
```

4. VALUES AND TYPES

The inputs and outputs of SISAL expressions and functions are values. The entire collection of values that may be presented to or produced by SISAL programs is the value domain of SISAL. The value domain is subdivided into disjoint subdomains that are the data types of SISAL. There are basic types which include the familiar scalar values of computation; structured types in the form of arrays, streams, and records as defined by the language user in terms of simpler data types; and discriminated union types.

4.1. Type specifications

A type specification in SISAL is a syntactic construct that specifies a data type.

Syntax:

```
type-spec          ::=  basic-type-spec
                      |  compound-type-spec
                      |  type-name

basic-type-spec    ::=  boolean
                      |  character
                      |  double_real
                      |  integer
                      |  null
                      |  real

compound-type-spec ::=  array [ type-spec ]
                      |  stream [ type-spec ]
                      |  record [ field-spec [ ; field-spec ] ... [ ; ] ]
                      |  union [ tag-spec [ ; tag-spec ] ... [ ; ] ]

field-spec         ::=  field-name [ , field-name ] ... : type-spec

tag-spec           ::=  tag-name [ , tag-name ] ... [ : type-spec ]

field-name         ::=  name

tag-name           ::=  name

type-name          ::=  name
```

For a basic type, the specification is simply the name of the type. For a compound type, the specification consists of the name of the compound type followed by the necessary additional information within square brackets. Any type

name used as a type specification must be defined by a type definition (see Section 4.6).

4.1.1. Array type

The array type specification gives the type of the elements of the array (also called the base type of the array).

Examples:

```
array[ integer ]
array[ array[ real ] ]
```

4.1.2. Stream type

The stream type specification gives the type of the elements of the stream (again known as the base type of the stream).

Examples:

```
stream[ integer ]
stream[ record[ X : real; Y : integer ] ]
```

4.1.3. Record type

The record type specification gives the field names and the type associated with each field. The field names used within any record specification must be distinct. Where several field names are listed with one type, the fields are all of that type.

Examples:

```
record[ I, J : integer; Temp: real ]

record[ I : record[ X : array[ boolean ]; Y : character ];
          Temp: real ]
```

A name (that is not a reserved word) may be used as a field name and as some other name without conflict, since it is interpreted as a field name only in the record type specification and in the record operations. The same field name may be used in several record types without conflict.

4.1.4. Union type

The discriminated union type specification gives the tags and the type

associated with each tag. The tag names must be distinct. Where several tag names are listed with one type, the tags all indicate that type. If the colon and following type specification are omitted, the **null** type is assumed.

Examples:

```
union[ Up, Down, Left, Right ]
union[ Fix : integer; Flo : real ]
union[     This          : array[ integer ];
           That,
           The_Other     : record[ C : real; D : boolean ] ]
```

In the first example we have effectively created an enumeration type, as is done in Pascal. However, their use is limited to the tagcase expression (as opposed to the pred and succ functions of Pascal).

As in the case of field names, a tag name may coincide with any other name without conflict, and the same tag name may be used in several union types without conflict.

4.2. Value domains

Each data type is a domain of values as described below. As will be seen, each data type includes proper elements, and an error element which occurs as the result of an expression when the computation of a proper value of the type is impossible. Each data type is further characterized by the set of operations that may be used to create and transform values of the type. The operations for each data type of SISAL are defined in Section 5, as are conversion operations that convert values of one type into values of another.

4.3. Error values

A simple error handling approach has been adopted in SISAL. In SISAL, all primitive operations are well defined for all (type correct) inputs. SISAL's approach adds one error value to each type (both primitive and user-defined types).

The full name of an error value consists of "**error**" followed by a type specification enclosed in brackets (e.g., **error[real]**). The type specification is required because every value must have a unique type. For example, the value **error[real]** is different from **error[integer]**.

The value **error[** type-specification **]** occurs when the result of an arithmetic or control operation is needed, but cannot be produced. General circumstances that can cause **error** to be returned are:

(1) Array access outside the current array bounds.

(2) Stream access beyond the end of the stream.

(3) In many primitive operations, when an input is **error**, the result will also be **error**. Major exceptions include: the **is error** function which is used to detect **error** and accessing valid elements of erroneous arrays.

(4) When the predicate of an **if** expression produces an error value, all results of the expression are **error** of the appropriate type. (Similar results occur in **tagcase** expressions).

(5) When the predicate of a **for** expression produces an error value, the expression terminates immediately. All partially-constructed array and stream results are returned as **error** values, but the correct portions of these structured objects may be accessed without producing an error value. All other results are **error** of the appropriate type.

(6) When a **for** range expression uses the **dot** product option, and the ranges have differing lengths, the shorter ranges will be padded with **error** values to match the longest length.

Type-specific circumstances that yield **error** are discussed in the sections below that discuss each type.

Generally, predefined SISAL functions (except for the error test function described in section 5.1) map error-valued inputs to the appropriate error values on output. If a SISAL-defined function receives an error-valued input, the outputs of that function will be the error value (adjusted to match the specified output types for that function). Exceptions to this rule are listed with each specific function.

4.4. Basic types

4.4.1. The NULL type

proper elements: **nil**

error element: **error**[**null**]

The **null** type occurs in a distinguished union (**union**) type where in one or more alternatives no data value is required.

Example:

```
type List = union[ NoMore : null;
                   ListElement : record [ Data : integer; Next : List ] ]
```

4.4.2. The BOOLEAN type

proper elements: **true, false**

error element: **error**[**boolean**]

4.4.3. The INTEGER type

proper elements: The integers between some implementation dependent limits.

error element: **error[integer]**

Integer-specific causes of **error** include values that are:

(1) Too large or small to be represented by the implementation.
(2) Possibly out of the implemented domain, but possibly in it (e.g., **error[integer]** - 1).
(3) The result of a division or modulus operation with a zero divisor.

4.4.4. The REAL type

proper elements: Floating point representations of real numbers including zero, with some exponent range and number of significant mantissa digits that are implementation dependent.

error element: **error[real]**

Real-specific causes of **error** include values that are:

(1) Too large (positive or negative) to represent in the implementation.
(2) Too small (positive or negative), but non-zero, to represent in the implementation.
(3) Possibly out of the implemented domain, but possibly in it (e.g., **error[real]** - 5.0).
(4) The result of an attempted division by zero.

4.4.5. The DOUBLE_REAL type

proper elements: Floating point representations of real numbers including zero, with some exponent range and number of significant mantissa digits that are implementation dependent. Either or both the exponent range and the number of significant mantissa digits are assumed to be greater than or equal to the corresponding **real** domains.

error element: **error[double_real]**

Double_real-specific causes of **error** include values that are:

(1) Too large (positive or negative) to represent in the implementation.
(2) Too small (positive or negative), but non-zero, to represent in the implementation.
(3) Possibly out of the implemented domain, but possibly in it (e.g., **error[double_real]** - 5D0).

(4) The result of an attempted division by zero.

Constants of **double_real** type are represented in a manner similar to those of **real** type. **double_real** constants must contain an exponent part, with the characters 'D' or 'd' replacing the 'E' and 'e' found in **real** constants.

4.4.6. The CHARACTER type

proper elements: The 128 characters of the ASCII character set.

error element: **error**[**character**]

Character-specific causes of the value **error** include:

(1) **Character** applied to an argument greater than 127 or less than 0.

4.5. Compound types

4.5.1. ARRAY types

For each data type defined by some SISAL type specification T, an array type may be defined by the type specification **array**[T].

proper elements: A proper array value of **array**[T] consists of two components:

(1) A range (LO, HI) where LO and HI are integers. LO and HI are inclusive bounds on the array indices. If HI < LO the array has no elements.

(2) A sequence of HI - LO + 1 elements of type T. The sequence cannot be sparse, that is, there are no indices in the sequence for which the corresponding element of type T does not exist.

error element: Every array type **array**[T] includes **error**[**array**[T]]

Error-valued arrays (i.e., arrays that cause the **is error** function to return **true**) indicate that some computational problem prevented the complete definition of the array. Two general conditions can produce erroneous arrays: (1) an array has a bad lower bound and hence there can be no correspondence between array indices and values held in the array, and (2) some operation may begin to build a valid array, but stop prematurely (e.g., a loop producing an array may have its termination test yield the **error** value, or a catenate operation adding an erroneous array to a good array). In the latter case, if portions of the array were correctly built prior to the problem arising, these portions

will be accessible by normal subscripting means, *even though the array has the value* **error**. These values can also be accessed through various versions of the **for** expression. See sections 5.7 and 7.4 of this manual for details.

If an array has a bad lower bound, the values associated with that array are not completely lost. For example, assume Aerr is a result of trying to shift the origin of a valid array (Aok) to a base index whose value is **error**. In this type of case, Aerr will not have any accessible elements, but the element values will not be lost. If a third array (Aok2) is built by shifting the origin of Aerr to a legal value, this array will have all of the elements from Aok accessible. We will refer to the values in Aerr as *hidden values*.

These two types of error conditions can occur simultaneously in a single array. In the previous example, if Aok is erroneous with accessible values, these values will be hidden in Aerr, and accessible again in Aok2. However, Aok2 will still be an erroneous array.

4.5.2. STREAM types

For each data type defined by some SISAL type specification T, a stream type may be defined by the type specification **stream**[T].

proper elements: A proper stream value of **stream**[T] consists of two components:

(1) A range (1, HI) where 1 and HI are integers and HI >= 0. These are inclusive bounds on the defined elements. If HI = 0, the stream has no elements.

(2) A sequence of HI elements of type T. The sequence cannot be sparse, that is, there are no integers in the sequence for which the corresponding element of type T does not exist.

error element: Every stream type **stream**[T] includes **error**[**stream**[T]]

Error-valued streams indicate that some computational problem prevented the complete construction of the stream. If portions of the stream were correctly built prior to the problem arising, these portions will be accessible by normal referencing means, *even though the stream has the value* **error**. These values can also be accessed through various versions of the **for** expression. See sections 5.8 and 7.4 of this manual for details.

4.5.3. RECORD types

If t1,...,tk are SISAL type specifications and n1,...,nk are distinct names, then **record**[n1 : t1; ...; nk : tk] specifies a record type T.

proper elements: Each proper value of the record type is a set of k pairs: { (n1, v1), ..., (nk, vk) } where each vi is an element of ti.

error element: **error**[T]

4.5.4. UNION types

Each element of a union type is an element of one of several constituent types, accompanied by a tag which indicates the constituent type from which the element was taken. If t1,...,tk are type specifications, and n1,...,nk are distinct names, then **union**[n1 : t1;...; nk : tk] specifies a union type T.

proper elements: Each proper element of the union type is a pair (ni, vi) where 1 <= i <= k and vi is an element of ti.

error element: **error**[T]

4.6. Type definitions

Syntax:

type-def ::= **type** type-name = type-spec

type-name ::= name

A function definition may contain type definitions which specify programmer-named types used in the function. Similarly, a compilation unit may contain type definitions used in any of the functions or other type definitions that are defined by the compilation unit. Each type definition specifies that a type name denotes the type represented by the given type specification. The type specification part of a type definition may contain type names defined in the same or other definitions. Type definitions may be used to construct data types composed of array, stream, record, or union structures of unlimited depth.

Example:

```
type Stack = union[ Empty : null;
                    Element : record[ Value : real; Rest : Stack ] ]
```

The type name Stack is declared by its appearance on the left hand side of the equals sign in the above definition. The name of a defined type may be used anywhere that a type specification is permitted, e.g., as the type parameter for constants such as **error**[type-spec]. A name may be used as a type name and as any other kind of name without conflict, since it is interpreted as a type name only in well defined contexts.

4.7. Conformance of type specifications

Type checking is performed by the SISAL translator by testing that the type of each expression or subexpression matches the type required by the context in which it appears. The type of an expression or subexpression is determined by its composition from operators and elementary terms as described in Sections 5 and 6. The type must match the type required by its context: an argument to a function must match the argument type indicated in the function's definition, and an expression on the right hand side of a definition (see Section 7.2) must match the declared type of the name on the left hand side (if one was given).

The necessary test determines if two type specifications conform, that is, if they denote the same type. Two basic type specifications conform if they are the same. Two array or stream specifications conform if their base types conform. Two record or union type specifications conform if their correspondingly named component types or constituent types conform; the order in which they are listed must be the same. A defined type name conforms to the type specification appearing on the right hand side of its definition.

A compound type specification may be visualized as a tree whose nodes are labeled **array, stream, record,** or **union,** whose arcs from **record** or **union** nodes are labeled with field or tag names, and whose leaves are basic types. Conformance can be formulated in terms of this characterization: two type specifications conform if their trees are identical. If a type specification uses recursion, this tree is infinite; two such specifications conform if these infinite trees are identical.

Examples -- assume the following type definitions:

```
type    Num   = real
type    Stack = union[ Empty : null; Element : Item ]
type    Item  = record[ Value : real; Rest : Stack ]
```

Then the following pairs of type specifications conform:

```
real                    (A defined type exactly conforms to
Num                     the type that it is defined to be.)

Item                    (A type name conforms to its definition.)
record[ Value : real; Rest : Stack ]

union[ Empty : null; Element : record[ Value : real; Rest : Stack ] ]
Stack                   (The (infinite) trees implied by
                        these type specifications conform.)
```

5. OPERATIONS

In this section we specify the sets of operations applicable to each data type of SISAL. In the examples of notation, P and Q stand for boolean values, J and K for integers, X and Y for reals, C and D for characters, A and B for arrays, G and H for streams, R for records, U for union values, N for record field names and tag names, T for arbitrary types, and V for values of arbitrary type.

5.1. Error tests

operation	notation	functionality
test for error	**is error**(V)	any -> bool

The error test operation always return **true** or **false**, never an error value. It must be used for testing for errors instead of the equality operator (e.g., "X = **error**[**real**]"), since the latter returns **error**[**boolean**] when X is an error value.

5.2. Null operations

The null type is used to provide a case in a union type for which the value is irrelevant. There are no operations for this type except the error test **is error**.

5.3. Boolean operations

The boolean operations are the following:

operation	notation	functionality
and	P & Q	bool,bool->bool
or	P | Q	bool,bool->bool
not	~P	bool->bool
equal	P = Q	bool,bool->bool
not equal	P ~= Q	bool,bool->bool

The results of these functions when given error-valued inputs are as described in Section 4.3, except for the functions "and" and "or". If either input to "and" is **false**, the result is **false** (even if the other input is **error**). Otherwise,

an **error** input produces the result **error**. Likewise for "or", if either input is **true**, the result is **true** (even if the other input is **error**). Otherwise, an **error** input produces the result **error**. Also, the Error test described in Section 5.1 applies to Booleans.

5.4. Integer operations

The integer operations are the following:

operation	notation	functionality
addition	J + K	int,int ->int
subtraction	J - K	int,int ->int
multiplication	J * K	int,int ->int
division	J / K	int,int ->int
modulus	mod(J,K)	int,int ->int
exponentiation	exp(J,K)	int,int ->int
negation	- J	int -> int
magnitude	abs(J)	int -> int
maximum	max(J,K)	int,int ->int
minumum	min(J,K)	int,int ->int
equal	J = K	int,int -> bool
not equal	J ~= K	int,int -> bool
greater, less	J > K, J < K	int,int -> bool
greater/equal, less/equal	J >= K, J <= K	int,int -> bool

The results of these functions when given error-valued inputs are as described in Section 4.3. Also, the Error test described in Section 5.1 applies to integers.

The only two operations that require further clarification are division for integers and modulus. Their values are defined as follows:

J / K = I, where I is the result of computing J / K using real division and then truncating toward zero.

mod (J, K) = L, where L has the following properties:
0 <= | L | < | K | , sign(L) = sign(K),
and there exists an integer M such that
J = M * K + L.

The error value will result from an attempt to use zero as the divisor in the division or modulus operations or from the arithmetic operations if the result exceeds the range of numbers representable on the target computer.

5.5. Real operations

The real operations are the following:

operation	notation	functionality
addition	X + Y	real,real->real
subtraction	X - Y	real,real->real
multiplication	X * Y	real,real->real
division	X / Y	real,real->real
exponentiation	exp(X,Y)	real,real->real
exponentiation with integer	exp(X,J)	real,int->real
negation	- X	real->real
magnitude	abs(X)	real->real
maximum	max(X,Y)	real,real->real
minimum	min(X,Y)	real,real->real
equal	X = Y	real,real->bool
not equal	X ~= Y	real,real->bool
greater, less	X > Y, X < Y	real,real->bool
greater/equal, less/equal	X >= Y, X <= Y	real,real->bool

The error value will result from an attempt to use zero as the divisor in a division operation, or if the result of an arithmetic operation exceeds the range of numbers representable on the target computer.

The results of these functions when given error-valued inputs are as described in Section 4.3. Also, the Error test described in Section 5.1 applies to reals.

The operations on values of **double_real** type are identical to those for values of **real** type.

5.6. Character operations

The character operations are the following:

operation	notation	functionality
equal	C = D	char,char->bool
not equal	C ~= D	char,char->bool
greater, less	C > D, C < D	char,char->bool
greater/equal, less/equal	C >= D, C <= D	char,char->bool

The results of these functions when given error-valued inputs are as described in Section 4.3. Also, the Error test described in Section 5.1 applies to characters.

5.7. Array operations

The operations for the array data type **array**[T] include creation of new arrays, selection, producing new array values by appending components to an array value, and combining arrays by concatenation. Recall that an array value consists of a range defined by a low index LO, a high index HI, and a sequence of HI-LO+1 elements of the given type.

operation	notation	functionality
create	**array** type-name [ ]	->array[T]
create by elements	**array** [type-name] [J : V]	int,T->array[T]
create/fill	array_fill(LO,HI,V)	int,int,T->array[T]
select	A [J]	array[T],int->T
replace	A [J : V]	array[T],int,T->array[T]
concatenate	A || B	array[T],array[T]->array[T]
index of highest	array_limh(A)	array[T]->int
index of lowest	array_liml(A)	array[T]->int

number of elements	array_size(A)	array[T]->int
number of elements	array_prefixsize(A)	array[T]->int
set bounds	array_adjust(A,LO,HI)	array[T],int,int->array[T]
extend high	array_addh(A,V)	array[T],T->array[T]
extend low	array_addl(A,V)	array[T],T->array[T]
remove high	array_remh(A)	array[T]->array[T]
remove low	array_reml(A)	array[T]->array[T]
set low limit	array_setl(A,LO)	array[T],int->array[T]

In general, the results of these functions when given error-valued inputs are as described in Section 4.3. All exceptions to this rule are described below for each operation. Also, the Error test described in Section 5.1 applies to arrays.

The following subsections give an informal semantics for the operations listed above. A more formal and precise semantics of these operations can be found in Appendix G of this manual. This section defines both the normal and error-handling treatment of arrays in SISAL.

5.7.1. Create

array type-name [ ]

This is an array of the indicated type, whose low index is one, high index is zero, and which therefore contains no elements. The type-name is mandatory. Note : The specified type name denotes the type of the **array** operation, and therefore must be an **array** type -- not the type of the component.

5.7.2. Create by elements

array [type-name] [J : V]

This returns an array of the indicated type with low and high indices both J, and one element V at index J. The type-name is optional, but if present, must conform to the type of V. This operation yields a proper array even if V is the error value. If J is an error value, then the result is an error array with V becoming a hidden value for that array.

Note : The specified type name denotes the type of the **array** operation, and therefore must be an **array** type -- not the type of the component.

There are abbreviated notations for compositions of select, replace, and create by elements operations to simplify construction of multiple element arrays and for operating on multi-dimensional arrays. See Section 6.4.

5.7.3. Create/fill

array_fill (LO,HI,V)

This creates an array with the given range and all elements equal to the given value. If LO > HI, the result is a valid empty array with a lower bound set to LO. If LO or HI are the error value, the result is an error array with no hidden values. This operation yields a proper array even if V is an error value.

Example:

array_fill(1,10,6)

is of type **array**[**integer**] with 10 elements, all equal to 6.

5.7.4. Select

A [J]

This operation yields the element of the array A at index J. If J is not within the range of the array, the result is **error**[T]. Otherwise, the result is whatever value is associated with the index value in the array, which may be an error value. This operation will access valid portions of erroneous arrays but it will not access hidden values.

5.7.5. Replace

A [J : V]

If A is a valid array and J lies within the bounds of A, this operation returns an array identical to A except that the element at index J has been replaced by value V. If A is in error, but has accessible elements, and J refers to a position within those accessible elements, then the result is an error array having the same accessible values as A, with the one change of V at position J. Otherwise, the result of the operation is **error**[**array**[T]] with no accessible or hidden values.

5.7.6. Concatenate

A || B

If A and B are valid arrays, this returns an array whose size is the sum of the sizes of A and B, formed by concatenating A and B. The low index of the result is the same as the low index of A, and the elements of A retain their original indices. The indices of B are shifted as necessary. In any error situations where B has no proper lower bound, the result is unaltered because that lower bound value has no bearing on this operation. If A has an improper lower bound, the result also has an improper lower bound. The actual values in the result depend on whether A and B have accessible values. If A is valid, but B is erroneous with accessible values, then the result is an array that is in error, but retains all of A values as accessible elements and has all of B's accessible values added on the end. If A is erroneous with accessible values, then the result is A. (This last case has some subtle implications if B is a valid array, see Appendix G for the details.)

5.7.7. Index of highest, lowest

array_limh(A), array_liml(A)

These functions return the high or low index of A, respectively. If A is in error, but has a valid lower bound, the low limit operation will return that lower bound.

5.7.8. Number of elements: good arrays only

array_size(A)

This function returns the size of an array in all cases where that size was not affected by any erroneous values. In addition to returning the proper value for all non-erroneous arrays, it also produces the correct value if the only erroneous aspect of the array is an invalid lower bound. For all other cases this function returns an error.

5.7.9. Number of elements: good or bad arrays

array_prefixsize(A)

This function always returns the number of accessible values in the array A, whether or not A is in error. If the lower bound of A is in error, it returns the number of elements that would become accessible if this bound were set to any legal value. In all cases this function returns a valid integer.

5.7.10. Set bounds

array_adjust(A,LO,HI)

This returns an array with range (LO,HI), containing the same data as A where possible. If LO is greater than array_liml(A) or HI is less than array_limh(A), some elements of A will be absent in the result. If LO is less than array_liml(A) or

HI is greater than array_limh(A), the result contains **error** [T] in the out-of-range positions. Any accessible values in A between LO and HI will be accessible in the result, even if A is erroneous. The resulting array will be erroneous only if the range includes inaccessible portions of A.

5.7.11. Extend high,low

array_addh(A,V), array_addl(A,V)

These return the array A with its high index increased by one or its low index decreased by one, and the given value V as the new element. This definition applies to all possible values for V, including error values. If A is erroneous with hidden values, the result is an error value with V added appropriately to the list of hidden values. In the array_addl operation, if A is an error value with accessible elements, the result is like A with V appended to the beginning of the array. In the array_addh operation, if A is an error value with accessible elements, the result is **not** appended as indicated. For a complete treatment of this last situation, please refer to Appendix G.

5.7.12. Remove high,low

array_remh(A), array_reml(A)

These return the array A with its high index decreased by one or its low index increased by one. An element of A is lost in the result. If an array A has size zero, the result is **error**[**array**[T]]. This definition also applies in the situations where A is erroneous. The reader should be warned that there are a few very special circumstances when an erroneous array with hidden or accessible values **will not** lose an element in an array_remh. For precise details of this situation, please refer to Appendix G.

5.7.13. Set low limit

array_setl(A,LO)

This adds LO - array_liml(A) to all element indices and to both components of the range, yielding an array similar to A but with the origin shifted. Its low index is LO. If LO is the error value, the result is an erroneous array having all elements of A as hidden values. If LO is a valid integer and A is erroneous with hidden values, then the result is a **valid** array with all of the hidden values in A being accessible in the result. If A is erroneous with accessible values, the result is still in error with the accessible values based from the new lower bound.

array_setl(**array**[2 : X, Y, Z], 5)

denotes the same value as

array[5 : X, Y, Z]

where the abbreviated notation is defined in Section 6.4.

5.8. Stream operations

The stream operations are the following:

operation	notation	functionality
create	**stream** type-name [ ]	->stream[T]
append	stream_append(G,V)	stream[T],T->stream[T]
select first	stream_first(G)	stream[T]->T
select all but first	stream_rest(G)	stream[T]->stream[T]
test for empty	stream_empty(G)	stream[T]->boolean
number of elements	stream_size(G)	stream[T]->integer
number of elements	stream_prefixsize(G)	stream[T]->integer
concatenate	G || H	stream[T],stream[T]->stream[T]

In general, the results of these functions when given error-valued inputs are as described in Section 4.3. All exceptions to this rule are described below for each operation. Also, the Error test described in Section 5.1 applies to streams.

5.8.1. Create

stream type-name [ ]

This is a stream of the indicated type which contains no elements. The type-name is mandatory. Note : the type-name is the type of the **stream** operation and therefore must be a stream type -- it is not the type of the stream's components.

5.8.2. Append

stream_append(G,V)

This operation returns a stream that is identical to the input stream except that the element V has been added to the end of the stream. If V is the error

value, it is appended to the stream G. If the stream G is an error value, then the result is G.

5.8.3. Select first element

stream_first(G)

This operation returns the first element of the stream. If the stream is an error value but has accessible elements, the first of those elements is the result. If the stream is empty, it returns **error**[T].

5.8.4. Select all but first element

stream_rest(G)

This operation returns a stream that is identical to the input stream, except that the first element has been removed. If the stream is in error with accessible values, this rule still applies. If the stream is empty it returns **error**[**stream**[T]].

5.8.5. Test for empty

stream_empty(G)

This operation returns **true** if the stream has no elements, otherwise **false**. If the stream is in error with accessible values, this rule still applies.

5.8.6. Number of elements: non-erroneous streams

stream_size(G)

This returns the number of elements in G. If G is error-valued, the result **error**.

5.8.7. Number of elements: all streams

stream_prefixsize(G)

This returns the number of elements in G. If G is error-valued, the result is the number of accessible elements of the stream.

5.8.8. Concatenate

G || H

This operation returns a new stream containing all the elements of G followed by all the elements of H. The order of all elements in G and H is preserved. If G is any error value, the result is G. If G is a non-erroneous stream but H is an error value, the result is an error-valued stream with the following accessible elements: all elements of G followed by any accessible elements of H.

There are abbreviated notations for composition of the create and append operations to simplify construction of multiple element streams. See Section 6.5.

5.9. Record operations

The record operations for a record type specified as T = **record**[N1 : T1; ...; Nk : Tk] are the following. N1,...,Nk are the field names, and T1,...,Tk are the corresponding types.

operation	notation	functionality
create	**record** [type-name] [N1 : V1; ...; Nk : Vk]	T1,...,Tk->T
select, 1<=i<=k	R . Ni	T->Ti
replace, 1<=i<=k	R **replace** [Ni : V]	T,Ti->T

In general, the results of these functions when given error-valued inputs are as described in Section 4.3. All exceptions to this rule are described below for each operation. Also, the Error test described in Section 5.1 applies to records.

5.9.1. Create

record [type-name] [N1 : V1; ...; Nk : Vk]

This builds a record value { (N1, V1), ..., (Nk, Vk) }. All of the field names associated with the type of record being constructed must appear in the list, though some may appear with error values. The type-name is optional, but if present, the record value must conform to the designated type.

5.9.2. Select

R . N

This returns the value of the named field, that is, Vi if N = Ni.

5.9.3. Replace

R **replace** [N : V]

This returns a record similar to R except that the N-field value is changed to V. This result applies whether or not V is an error value. If R is an error value, the result is a non-erroneous record with V as the value in field N, and all other fields having values of **error** of the appropriate type.

Abbreviated notations are provided for compound selectors and multiple values in replace operations. See Section 6.6.

5.10. Operations for union types

The basic operations for a union type specified as T = **union**[N1 : T1; ...; Nk : Tk] are a create operation and a test of a tag. The tagcase control structure explained in Section 7.3 is the mechanism for accessing constituent values from a value of union type. In the following, N1,...,Nk are the tag names, and T1,...,Tk are the corresponding constituent types.

operation	notation	functionality
create, 1<=i<=k	**union** type-name [Ni [: V]]	Ti->T
tag test, 1<=i<=k	**is** Ni(U)	T->bool

In general, the results of these functions when given error-valued inputs are as described in Section 4.3. All exceptions to this rule are described below for each operation. Also, the Error test described in Section 5.1 applies to unions.

5.10.1. Create

union type-name [N [: V]]

The result of this operation is a union value of the indicated type with a tag N and an element V. This definition applies whether or not V is an error value. If the colon and the element V are not provided, V is assumed to be **nil**. The type of V must conform to the type corresponding to tag N.

5.8.8. Concatenate

G || H

This operation returns a new stream containing all the elements of G followed by all the elements of H. The order of all elements in G and H is preserved. If G is any error value, the result is G. If G is a non-erroneous stream but H is an error value, the result is an error-valued stream with the following accessible elements: all elements of G followed by any accessible elements of H.

There are abbreviated notations for composition of the create and append operations to simplify construction of multiple element streams. See Section 6.5.

5.9. Record operations

The record operations for a record type specified as T = **record**[N1 : T1; ...; Nk : Tk] are the following. N1,...,Nk are the field names, and T1,...,Tk are the corresponding types.

operation	notation	functionality
create	**record** [type-name] [N1 : V1; ...; Nk : Vk]	T1,...,Tk->T
select, 1<=i<=k	R . Ni	T->Ti
replace, 1<=i<=k	R **replace** [Ni : V]	T,Ti->T

In general, the results of these functions when given error-valued inputs are as described in Section 4.3. All exceptions to this rule are described below for each operation. Also, the Error test described in Section 5.1 applies to records.

5.9.1. Create

record [type-name] [N1 : V1; ...; Nk : Vk]

This builds a record value { (N1, V1), ..., (Nk, Vk) }. All of the field names associated with the type of record being constructed must appear in the list, though some may appear with error values. The type-name is optional, but if present, the record value must conform to the designated type.

5.9.2. Select

R . N

This returns the value of the named field, that is, Vi if N = Ni.

5.9.3. Replace

R **replace** [N : V]

This returns a record similar to R except that the N-field value is changed to V. This result applies whether or not V is an error value. If R is an error value, the result is a non-erroneous record with V as the value in field N, and all other fields having values of **error** of the appropriate type.

Abbreviated notations are provided for compound selectors and multiple values in replace operations. See Section 6.6.

5.10. Operations for union types

The basic operations for a union type specified as T = **union**[N1 : T1; ...; Nk : Tk] are a create operation and a test of a tag. The tagcase control structure explained in Section 7.3 is the mechanism for accessing constituent values from a value of union type. In the following, N1,...,Nk are the tag names, and T1,...,Tk are the corresponding constituent types.

operation	notation	functionality
create, 1<=i<=k	**union** type-name [Ni [: V]]	Ti->T
tag test, 1<=i<=k	**is** Ni(U)	T->bool

In general, the results of these functions when given error-valued inputs are as described in Section 4.3. All exceptions to this rule are described below for each operation. Also, the Error test described in Section 5.1 applies to unions.

5.10.1. Create

union type-name [N [: V]]

The result of this operation is a union value of the indicated type with a tag N and an element V. This definition applies whether or not V is an error value. If the colon and the element V are not provided, V is assumed to be **nil**. The type of V must conform to the type corresponding to tag N.

5.10.2. Tag test

is N(U)

The result of this operation is **true** if U was created with a tag N and value V. If U is **error**[T] the result is as specified in Section 4.3 . Otherwise, the result is **false**.

5.11. Type conversion operations

Type conversion operations are provided between integers and reals and between integers and characters. The operations use the basic type names as if they were function names. In all cases, the rules for handling erroneous inputs follow the general guidelines in Section 4.3.

operation	notation	functionality
real-to-integer	floor(X)	real->int, double_real->int
	integer(X)	real->int, double_real->int
	trunc(X)	real->int, double_real->int
integer-to-real	**real**(J)	int->real
	double_real(J)	int->double_real
real-to-real	**real**(X)	double_real->real
	double_real(X)	real->double_real
character-to-integer	**integer**(C)	char->int
integer-to-character	**character**(J)	int->char

5.11.1. Floor(X)

If X is larger in magnitude than is representable as a proper element of **integer**, the result is **error**[**integer**]. Otherwise, the result is the largest integer not greater than X.

5.11.2. Integer(X)

If X is larger in magnitude than is representable as a proper element of **integer**, the result is **error**[**integer**]. Otherwise, the result is obtained by adding .5 to X and then applying **floor**.

5.11.3. Trunc(X)

If X is larger in magnitude than is representable as a proper element of **integer**, the result is **error**[**integer**]. Otherwise, the result is obtained by deleting any non-integral portion of X.

5.11.4. Real(J), Double_real(J)

All proper values of J are converted to the corresponding reals. The conversion to **real** or **double_real** is rounded.

5.11.5. Real(X)

All proper values of X are converted to the corresponding reals. The conversion from **double_real** to **real** is rounded.

5.11.6. Double_real(X)

All proper values of X are converted to the corresponding double reals.

5.11.7. Integer(C)

This operation yields the ASCII code for the character C.

5.11.8. Character(J)

This operation is the inverse of **integer**(C). If the value of J does not produce an ASCII character, the result is **error**[**character**].

5.12. Type correctness of operations

In SISAL the type of value produced by each expression can be determined by the translator from the properties of the operations as specified in this section. An operation in a program is type correct if and only if the types of its argument expressions conform with the argument types specified for the operation. Note that for each operator the types of the results are determined when the types of the arguments are known.

6. CONSTANTS, VALUE NAMES, AND EXPRESSIONS

An expression is the basic syntactic unit denoting a tuple of values of some types. The arity of an expression is the size of the tuple of values it denotes. Two expressions are said to conform if they have the same arity and the corresponding values are of the same type. The design of the SISAL language is such that the arity and types of an expression, and hence the conformity of two expressions, may be determined by inspection of the program. The simplest type of expression of arity one is a constant, a value name, or an operation applied to other expressions of arity one. The simplest type of expression of higher arity is a series of expressions of arity one separated by commas.

6.1. Constants

A constant is a lexical unit of arity one whose value and type are manifest from its form. The syntax for constants follows.

```
constant    ::=  false
             |   nil
             |   true
             |   integer-number
             |   real-number
             |   character-constant
             |   character-string-constant
             |   error [ type-spec ]
```

The value **error**[type-spec] denotes an error value of the type indicated in the type-spec. For example, **error**[**array**[**integer**]] denotes the undefined value of type **array**[**integer**]. This constant exists for all types, including array, stream, record, and union types. The remaining constants for each data type are as follows:

The only constant of the **null** type is the reserved word **nil**.

The constants of the **boolean** type are the reserved words **true** and **false**.

The constants of the **integer** and **real** types are integer and real numbers, the formats of which are given in Section 3.

The constants of the **character** type are the ASCII characters enclosed in single quotes, as described in section 3.

A character string enclosed in double quotes is a constant of type **array**[**character**] containing the individual characters of the string as elements. The first character is at index one.

There are no other array, stream, record, or union constants, but the various creation operations may be used with constant arguments to denote "constant" arrays, streams, records, or union elements.

Examples:

```
array[ 1 : 1,2,3,4,5 ]      % array constant, see Section 6.4
stream[ 1,2,3,4,5 ]         % stream constant, see Section 6.5
record[ A : 6; B : 7.3 ]    % record constant
union T [ A : 8 ]           % constant of union type T, tag A
```

6.2. Value names

A value name is a name which denotes a single computed value of a specified type. Every value name is introduced either in the header of a function definition (if the value name is a formal argument of the function being defined) or in a program construct such as a **let** block or a **for** block. In either case, each value name has a scope and a type, and has a unique value of that type for each instantiation during execution of the function or block with which the value name is associated. The scope of a value name is the region of program text in which a reference to the value name denotes its value. The scope and type of any value name may be determined by inspection of the program construct that introduces it. Its value of course depends on the values present during the particular instantiation of the function or block.

The scope of a value name introduced as a formal argument of a function is the entire function definition, less any inner scopes that re-introduce the same value name. The type of such a value name is given by a type declaration in the function header. Its value is the value of the corresponding argument for the relevant invocation of the function. See Sections 8.4 and 8.5 for a more complete discussion.

Example:

```
function F ( X : integer returns real)
<expression>
end function
```

An appearance of the value name X in the expression denotes the value of the argument with which F was invoked. Its type is **integer**.

The scope of a value name introduced in a program construct such as a **let** or **for** block is some region of the construct that depends on the nature of the construct, less any inner scopes that reintroduce the same value name. The manner in which the type and value of the value name are established depends on the form of the construct.

Example:

```
let
    X       : real  := 3.0;
    <other declarations and/or definitions>;
in
    <expression>
end let
```

The scope of X is the entire block, including the expression after **in**, less any

inner scopes that re-introduce X. Its type is **real**; its value is 3.0. The **let** construct is described in Section 7.2. If this block had appeared within the scope of an X introduced by some outer construct, that X, with its value and type, would disappear within this **let** block.

6.3. Expressions

Expressions are built out of smaller expressions by means of operation symbols.

Syntax:

```
expression        ::=  simple-expression [ , simple-expression ] ...

simple-expression ::=  primary [ binary-op primary ] ...

unary-op          ::=  + | - | ~

binary-op         ::=  < | <= | > | >= | = | ~=
                  |    + | - | ||
                  |    * | / | &
                  |    ||

primary           ::=  constant                 (arity 1)
                  |    unary-op primary         (arity 1)
                  |    old value-name           (arity 1)
                  |    value-name               (arity 1)
                  |    ( expression )           (arity of expression
                  |                             in parentheses)
                  |    invocation               (arity is the number of
                  |                             values returned)
                  |    array-ref                (arity 1)
                  |    array-generator          (arity 1)
                  |    stream-generator         (arity 1)
                  |    record-ref               (arity 1)
                  |    record-generator         (arity 1)
                  |    union-test               (arity 1)
                  |    union-generator          (arity 1)
                  |    error-test               (arity 1)
                  |    prefix-operation         (arity 1)
                  |
                  |    conditional-exp          ( These four expressions
                  |    let-in-exp                 are described in
                  |    tagcase-exp                Section 7. They have
                  |    iteration-exp              arbitrary arity. )
```

value-name ::= name

In an invocation, the arity of the expression in parentheses must be equal to the number of arguments required by the function.

invocation ::= function-name ([expression])

array-ref ::= primary [expression]

array-generator ::= **array** type-name [ ]
| **array** [type-name] [expr-pair]
| primary [expr-pair [; expr-pair] ... [;]]

expr-pair ::= expression : expression

stream-generator ::= **stream** type-name [ ]
| **stream** [type-name] [expression]

record-ref ::= primary . field-name

record-generator ::= **record** [type-name] [field-def [; field-def] ... [;]]
| primary **replace** [field : expression
[; field : expression] ... [;]]

field-def ::= field-name : expression

field ::= field-name [. field-name] ...

union-test ::= **is** tag-name (expression)

union-generator ::= **union** type-name [tag-name [: expression]]

error-test ::= **is error** (expression)

field-name ::= name
tag-name ::= name

prefix-operation ::= prefix-name (expression)

prefix-name ::= **character** (arity 1)
| **double_real** (arity 1)
| **integer** (arity 1)
| **real** (arity 1)

Operators obey the customary precedence rules, from highest to lowest:

unary arithmetic	+	-				
multiplicative	*	/				
binary additive	+	-				
concatenate	||					
relational	<	<=	>	>=	~=	=
unary boolean	~					
conjunction	&					
disjunction	|					

Examples of expressions of arity one:

```
A
true
3.7E-02
'%'
"XYZ" || array[ 1 : C ] || "PQR"
array[ ]
X > 2 & Z | Y                  % equivalent to ((X > 2) & Z) | Y
-X + 3 * B                     % equivalent to (-X) + ( 3 * B )
3 * (X + Y)
func(3 + X, Y)                 % if "func" returns one value
array[ 3 : Z ]
A[ 3 : Z ]
A[ 4, J ]                      % see Section 6.4 for this,
R.X.Y.Z                        % see Section 6.6 for this,
record[ A : P; B : Q ]         % this,
R replace[ A.X : P; B.Y : Q ] % and this
is A (U)
union T[ A : 3 ]
is error(X)
error[ real ]
if P then 4 else 5 end if              % see Section 7
```

6.4. Abbreviations for array operations

The syntax provides abbreviated forms for the select, replace, and create by elements operations, to allow convenient array creation and handling of multi-dimensional arrays.

Since multi-dimensional arrays are represented as arrays of arrays, the straightforward way to select an element is with an expression such as

```
A[J][K][L]
```

This may be written

```
A[J, K, L]
```

The expression within brackets has arity three.

The replace operation can be used for multi-dimensional arrays by using an expression of arity greater than one for the subscripts. Thus:

```
A[J, K, L : V]
```

is equivalent to

```
A[J : A[J][K : A[J, K][L : V]]]
```

that is, A with its J,K,L element replaced by V.

Replace operations may be composed by writing the J:V pairs in sequence within the brackets, separated by semicolons.

```
A[J1 : V1; J2 : V2; ...; JN : VN]
```

is equivalent to:

```
A[J1 : V1][J2 : V2] ...[JN : VN]
```

where, as noted below, Ji and/or Vi may be expressions of any arity.

Several values may be replaced at consecutive indices by using an expression of arity greater than one.

```
A[J : V, W, X]
```

is equivalent to:

```
A[J : V; J+1 : W; J+2 : X]
```

If multi-dimensional arrays are being used, the last index is the one that varies when multiple data items are present.

```
A[J,K,L : V,W,X]
```

is equivalent to:

```
A[J,K,L : V; J,K,L+1 : W; J,K,L+2 : X]
```

These expressions need not be constructed by listing expressions of arity one separated by commas. Other forms of expressions with higher arity will be described in Section 6.7. For example:

```
A[J : TRIPLE(X,Y,Z)]
```

fills in indices J, J+1, and J+2 if TRIPLE is a function returning three values.

All of the abbreviations permissible for the replace operation are permissible for the create by elements operation, with one exception. Array creates cannot be composed by writing a sequence of index-value pairs; such creates can only specify the base index of the array followed by a list of the array values.

Examples:

```
array [3 : X,Y,Z]
```

is an array with range (3,5), and elements X, Y, and Z.

```
array [1 : A]
```

is a "singleton" array with low and high indices both one.

6.5. Abbreviations for stream operations

The append and create operations may be composed. Assuming the type declaration

type Integer_stream = **stream**[**integer**]

the expression

stream[2, 3, 4, 5]

is equivalent to:

```
stream_append(
        stream_append(
                stream_append(
                        stream_append(
                                stream Integer_stream [],
                                2),
                        3),
                4),
        5)
```

6.6. Abbreviations for record operations

There are abbreviated forms for the replace operation to allow convenient handling of compound selectors and multiple data elements.

Accessing records with compound selectors is performed in the straightforward way:

R.A.B.C

Compound selectors may be used in replace operations by writing the field names separated by periods:

R **replace** [A.B.C : V]

is equivalent to

R **replace** [A : R.A **replace** [B : R.A.B **replace** [C : V]]]

that is, R with its A.B.C subcomponent replaced by V.

Replace operations may be composed by writing the N:V pairs in sequence within the brackets, separated by semicolons.

R **replace** [A : V; B : W; C.D : X]

is equivalent to

((R **replace** [A : V]) **replace** [B : W]) **replace** [C,D : X]

6.7. Expressions of higher arity

The program structures provided in SISAL for conditional computation and iteration are expressions of arbitrary arity, and are described in Section 7. Such expressions, or function invocations, may occur in program text in places that require a tuple of values of specified types: the argument list of an operation or function invocation, the body of a function definition, a list of array indices or elements in an array operation, the list of elements in a stream generator, or in building the program structures presented in Section 7.

6.8. Function invocations

A function invocation consists of the name of the function followed by an argument list within parentheses. The argument list is an expression, whose arity and types conform to the arguments required by the function. This information is given in the header of the function definition (see Section 8). The argument list is usually written as a series of expressions of arity one separated by commas, but it may be any expression. If a function has no input parameters, its invocation must still have the parentheses following the function name--i.e., a null argument list.

A function invocation is itself an expression whose arity and types are the number and types of the values returned by the function, which information also appears in expressions with complete generality, such as an argument to arithmetic, array, stream, and record operations. An invocation that returns several values may only be used where expressions of higher arity are permitted.

In the following examples, Single, Double, and Triple each take 3 arguments and return 1, 2, and 3 values, respectively:

```
K := 3 + ( Z * Single(X+1, 3, Single(X + 2, 4, W)) )
```

In the following example, if P is false, F and G are defined to be Double(X, Y, Z) while H is defined to be W:

```
F, G, H := if P     then  Triple(X, Y, Z)
                    else  Double(X, Y, Z), W
                    end if
```

Since the argument list for any function may be any expression, it may be a multiple-result function invocation or other program structure.

```
3 + Single( Triple(X, Y, Z) )

3 + Single( P, Double(X, Y, Z) )

4 + Single( if P then 4, 5 else Double(P, Q, R) end if, X)
```

The last example invokes Single with three arguments, of which the first two are either 4 and 5 or the two values returned by Double. The third argument to Single is always X.

7. PROGRAM STRUCTURES

The program structures described in this section are specific forms of expressions. If their arity is one, they may appear in arithmetic operations.

Example:

if P **then** X **else** Y **end if** + 3

This expression has value X + 3 or Y + 3, depending on P.

7.1. The IF construct

The conditional expression selects one of several expressions, depending on the values of boolean expressions.

Syntax:

conditional-exp ::= **if** expression **then** expression
[**elseif** expression **then** expression] ...
else expression
end if

The expressions following **if** and **elseif** are test expressions. Their arity must be one and their type **boolean**. The expressions following **then** and **else** are the arms. They must conform to each other, and the entire construct conforms to the arms.

The entire construct is an expression whose tuple of values is that of the first arm whose test expression is **true**, or the final arm if all test expressions are **false**. If any test expression needed to evaluate the construct is an error value, the value of the entire construct is a tuple of **error** values of the appropriate types. (If a test expression has value **true**, later test expressions are not needed and may have error values without affecting evaluation of the construct).

The **if** construct introduces no value names. All value name scopes pass into an **if** construct. If the scope of a value name includes an **if** construct, it includes all of the expressions of that construct, so that the value name may be used anywhere inside the conditional construct.

7.2. The LET construct

The purpose of the **let** construct is to introduce one or more value names, define their values, and evaluate an expression within their scope (that is, making use of their defined values).

Syntax:

let-in-exp	::=	**let** decldef-part **in** expression **end let**
decldef-part	::=	decldef [; decldef] ... [;]
decldef	::=	decl | def | decl [, decl] ... := expression
decl	::=	value-name [, value-name] ... : type-spec
def	::=	value-name [, value-name] ... := expression

Every value name introduced in a **let** block may be declared at most once and defined exactly once in that block. The declaration may be part of the definition, or it may be by itself preceding the definition. A value name need not be declared.

Examples:

```
X : integer;              % declaration
Y : real := 4.7 + Q;      % declaration as part of definition
W := 20.0;                % definition without type declaration
```

The declaration of a value name (if given) must precede or be part of its definition. Each value name must be defined before it is used (on the right hand side of another definition). Declarations and definitions may be mixed in any order as long as these requirements are met.

Several value names may be declared at once. The following phrase declares all three names to be real.

```
X, Y, Z, : real;
```

Several value names may be defined at once. The number and types of the names must conform to the arity and types of the expression on the right hand side.

```
X, Y, Z := 1.0, 2.0, 3.0;
P, Q, R := Triple(X, Y, Z);
```

Several value names may be declared and defined at once. In this case, each of a group of value names preceding a type specification are declared to be of that type.

```
X : integer, Y, Z : real := 3, 4.0, 5.0;
```

This declares X to be an **integer**, and both Y and Z to be **real**.

The declarations, definitions, and combined declarations and definitions are separated by semicolons.

The scope of each value name introduced in a **let** block is the entire block less any inner constructs that reintroduce the same value name. However, a value name must not be used in the definitions preceding its own definition.

All scopes for value names not introduced in a given **let** block pass into that block. Hence, if the scope of a value name (introduced by an outer construct) includes a **let** block and that value name is not reintroduced, it may be referred to freely within the block.

Example:

```
let
   X    : real;
   T    : real;

   T    := P + 3.7;
   X    := T + 2.4
in
   X * T
end let
```

In this example, the value of P is imported from the outer context. The scopes of T and X are both the entire block. A reference to X in the definition of T would be illegal because it is within the scope of X but does not follow the definition of X. The arity of this construct is one, and its type is **real**, because X * T has arity one and type **real**.

Since a value name may not be used until after it has been defined, and must be defined only once in a block, it may not appear in its own definition. Hence definitions such as

```
I := I + 1;
```

are never legal in **let** blocks (though a similar construct may occur in the iteration clauses of **for** blocks; see Section 7.4)

The expression following the word **in** is in the scope of all of the introduced value names, and hence can make use of their definitions. The entire **let** construct conforms to this expression.

7.3. The TAGCASE construct

This selects one of a number of expressions, depending on the tag of a **union** value, and extracts the constituent value.

Syntax:

tagcase-exp ::= **tagcase** [value-name :=] expression
tag-list : expression
[tag-list : expression] ...
[**otherwise** : expression]
end tagcase

tag-list ::= **tag** tag-name [, tag-name] ...

The entire construct is one expression whose values are those of the expression in the arm whose tag name matches the value of the test expression. If no match is found, and an **otherwise** clause is present, the arm following the word **otherwise** is used. All arms must conform to each other, and the entire construct conforms to the arms.

The expression following the word **tagcase** must be of arity one and of a **union** type. The tag names appearing in the arms of the construct must be tags of that **union** type. If they comprise all the tags of that type, the **otherwise** arm is not used; if not, the **otherwise** arm is required.

If a value name and ":=" appear after the word tagcase, that name is introduced for each arm of the construct except the otherwise arm. Its scope in each case is the expression in that arm, and its type is the constituent type indicated by the tag name for that arm. If an arm is evaluated (meaning that the tag of the test expression matches the tag name of the arm), the value name is defined to be the constituent value from the test expression. If the value name and ":=" do not appear, the constituent value is not made available inside the arms.

Example:

Let X be of type

union[A : **integer**; B : **array**[**integer**]; C : **real**; D : **boolean**]

If X has tag A and constituent value 3,

```
tagcase P := X
  tag A :
      P + 4
  tag B :
      P[6]
  otherwise :
      5
end tagcase
```

has value 7. The first arm is taken, and P (whose type is **integer** in that arm) is defined to be 3, the constituent value of X. If X has tag B and constituent value some array whose sixth element is 2, the value of the above construct is 2. In that case P is defined to be the array. If X has tag C or D, the construct has value 5. In that case the constituent value is not available, since the value name's scopes do not include the **otherwise** arm. (This is because the **otherwise** arm can encompass different constituent types, so the type of the value name could

not be determined.)

More than one tag name may share the same arm if they indicate the same type. In this case, the tag names are all listed, separated by commas, after the word **tag**.

Example:

Let X be of type

union[A : **integer**; B : **real**; C : **integer**]

Then the following is permissible.

```
tagcase P := X
  tag A, C :
      expression    (P is integer here)
  tag B :
      expression    (P is real here)
end tagcase
```

All scopes of value names other than the one appearing after the word **tagcase** pass into the construct. An outer scope for a value name with the same name as the one appearing after the word **tagcase** does not pass into the **tagcase** construct.

If the value of the test expression is an error value, the value of the entire construct is a tuple of **error** values of the appropriate types.

7.4. The FOR construct

There are two forms of this construct, one of which allows inner and/or outer (Cartesian) array and stream index products to be specified ('product' form), the other of which does not ('non-product' form). The non-product form performs sequential iteration in which one iteration cycle depends on the results of previous cycles. The product form is a special case of the non-product form that provides a more concise way to specify array and stream index and element sets. Both forms may generate a tuple of values of any type. The following text informally describes the semantics of the various **for** expression options. For more detailed information on the precise semantics, consult Appendix F. This appendix describes a set of transformations that convert SISAL programs with **for** expressions into equivalent SISAL programs that use recursion instead.

The scope of any value name defined in an outer construct passes into a **for** construct, unless that name is redefined within the **for** construct.

Syntax:

```
iteration-exp       ::=  for initial                    (non-product form)
                         decldef-part
                         iterator-terminator
                         returns return-exp-list
                         end for

                    |    for in-exp-list                (product form)
                         [ decldef-part ]
                         returns return-exp-list
                         end for

iterator-terminator ::=  iterator termination-test
                    |    termination-test iterator

iterator            ::=  repeat
                         iterator-body

termination-test    ::=  while expression
                    |    until expression

iterator-body       ::=  decldef-part

in-exp-list         ::=  in-exp
                    |    in-exp  dot in-exp [ dot in-exp ] ...
                    |    in-exp  cross in-exp [ cross in-exp ] ...

in-exp              ::=  value-name in expression [ at index-list ]

index-list          ::=  value-name [ , value-name ] ...

return-exp-list     ::=  return-clause ...

return-clause       ::=  [ old ] return-exp [ masking-clause ]

masking-clause      ::=  unless expression
                    |    when expression

return-exp          ::=  value of [ [ direction ] reduction-op ] expression
                    |    array of expression
                    |    stream of expression

direction           ::=  left
                    |    right
                    |    tree
```

reduction-op ::= **sum**
| **product**
| **least**
| **greatest**
| **catenate**

7.4.1. The non-product form of the FOR construct

7.4.1.1. Description

This expression computes its results based on iterative execution. It contains separate sections that handle initialization, repetitive action, termination testing, and result calculation. The general strategy is to introduce new (local) identifiers, called loop names, within the expression and on each loop pass to compute new values for the loop names based on their previous values. When the termination test is satisfied, the expression produces results that can be based on either the final values of each loop name, or on the sequence of values given to the loop names during their execution. The general structure of the expression is shown below.

Syntax:

iteration-exp ::= **for initial**
decldef-part
iterator-terminator
returns return-exp-list
end for

The scope of each loop name introduced in a **for** construct is the entire construct less any inner constructs that introduce the same value name. Loop names may be used in the definition of other loop names, so long as the 'definition before use' rule is observed. Each loop name must be defined before the end of the decldef-part.

The behavior of the non-product form is as follows. The loop names are initialized exactly once to the values indicated in the definitions appearing after the word **initial**. All loop names that will carry values from one loop pass to the next and all loop names that will be used in the **returns** clause must be given initial values. This initialization is treated as the first iteration of the expression. Subsequent execution depends on the location of the termination test. If the test appears prior to the **repeat** section, it executes prior to any rebindings defined in the iterator body. Otherwise, the test occurs after the rebindings. When the expression evaluates to **false** in a **while** test, or **true** in an **until** test, the iteration terminates.

The iterator body (denoted by **repeat**) conveys the new set of bindings to be established for the current loop iteration. A loop name's new value may require values computed in the previous iteration. Access to such values can be accomplished by using the **old** attribute (e.g., I := **old** I + 1 ;). Within the iterator body, use of a loop name without the **old** attribute indicates the current iteration's loop name value, and this name cannot be used until after its value

has been specified within the **repeat** section. Use of the **old** attribute indicates the value the loop name had on the previous loop pass, and it can be used throughout the iterator body. Any loop names that receive bindings in the iterator body but were not given initial values are treated as names local to each loop pass--they cannot carry information from one iteration to the next. Their values can, however, be accessed in termination tests that appear after the iteration body. Any loop names that are not given new values within the iterator body are assumed to be loop constants. All loop names can be redefined at most once within the iterator body.

At some point during loop execution each loop name's value must make a transition so that the value bound to a loop name becomes the value bound to the **old** loop name. This transition is defined to take place immediately prior to the execution of each iterator body. Therefore, termination tests appearing after the iteration body have access to both the current and **old** versions of the loop names and they have the same values as if they were to appear in the iteration body. Termination tests appearing before the iteration body cannot use the **old** attribute. Any references to loop names in such tests denote the most recent binding of values.

If an error occurs during evaluation of the boolean expression controlling the iteration, the iteration terminates and returns as its value a tuple of error values of the types specified by the return-exp-list. Array and stream results produced within a return expression (via **array of**, **stream of**, or **value of catenate**) have as accessible values any components correctly computed prior to the control error. If an error occurs during initial evaluation of the loop name values, or anywhere in the iterator-body, iteration continues using the resulting error values. If an error occurs during evaluation of a return-exp, the error value simply appears in the result tuple.

7.4.1.2. Result Values

The result value of a **for** construct is the tuple of values defined by the return-exp-list. As indicated earlier, the results of each **for** expression may depend on all values bound to the loop names during the course of a **for** evaluation. Each return-exp must contain some expression that describes a result to be produced. For purposes of semantic definition, assume that each such expression is evaluated after every cycle of the loop (counting initialization as one cycle). Therefore, each result has a sequence of values associated with it. SISAL provides two different means for altering these sequences (**old** and masking clauses) before they are used for producing results. SISAL then provides three different mechanisms for constructing the actual result values from the altered sequences: **value of**, **array of**, and **stream of**.

7.4.1.2.1. The OLD Modifier

Each result clause may be optionally prefixed with the modifier **old**. This option is permitted in lieu of allowing its arbitrary use within expressions appearing in the **returns** section. The **old** modifier alters the sequence of values used to produce results by removing the last element of the sequence, if it was produced by the last iteration.

7.4.1.2.2. The Masking clause Modifier

Each result clause may be optionally followed by a masking clause. A masking clause is a boolean expression preceded by either a **when** or **unless** indicator. These clauses act as filters to determine if specific sequence values should be taken out of the sequence prior to final result calculations. After each iteration cycle, the masking clauses are evaluated. If a **when** clause is **false** or if an **unless** clause is **true**, the corresponding expression value is dropped from the sequence. In cases where the boolean expression evaluates to an error value, the precise influence of the masking clause depends on the type of result being formed.

7.4.1.2.3. VALUE OF

The **value of** prefix signifies that the following phrase produces a single value in one of two ways.

If the **value of** is not followed by one of the reduction operators, the result produced by this clause is the last element of the sequence (as described above). Note that if there is a masking clause, the result will be the last value satisfied by the filter. If there is uncertainty about which value is the last one (because the last masking clause produced an error, the result is **error**.

If a reduction operator is in the clause, it means that all elements of the sequence will be combined using the reduction operator to produce the single value. In this case, the type of the sequence elements must be appropriate to the specific operator: **sum** and **product** accept **integer**, **real**, **double_real**, and **boolean**; **least** and **greatest** accept **integer**, **real**, and **double_real**; **catenate** accepts **array** and **stream** types. The following table summarizes the use of each of these operators.

operator name	reduction operation
sum	**integer**, **real**, or **double_real** addition (+), **boolean** or (|)
product	**integer**, **real**, or **double_real** multiplication(*), **boolean** and (&)
least	**integer**, **real**, or **double_real** minimum (**min**)
greatest	**integer**, **real**, or **double_real** maximum (**max**)
catenate	**array** and **stream** concatenation (||).

In the **value of** clauses that use a reduction operator, any masking clauses that produce errors (i.e., it cannot be determined whether or not to include a particular value in the sequence) causes an error value of the appropriate type to be inserted into the sequence. Hence, all of the reduction operators except **catenate** will yield the value **error** if any masking clause produces and error. In the case of **catenate**, the result will be an erroneous array or stream that has as accessible values all values computed prior to the first masking clause error. Note that all of the catenates after the first error are still performed. For the

full implications of this action, see appendix G.

A reduction operator may be preceded by one of the reserved words **left**, **right**, or **tree**. If it is not preceded by one of these, **left** is assumed. **right** may not be used with **catenate**. These three reserved words define the ordering imposed on the reduction operation. In the following, assume that a seven cycle iteration has produced the values 1,2,3,4,5,6 and 7, and that the **sum** operator is specified. **left** requires the values to be used in the order they are produced by the iteration, i.e., from left to right. The equivalent expression is

(((((1 + 2) + 3) + 4) + 5) + 6) + 7

Right requires the values to be used in the reverse order they are produced by the iteration. The equivalent expression is

1 + (2 + (3 + (4 + (5 + (6 + 7)))))

Tree forces the processor to perform the reduction in a logarithmic fashion. The equivalent expression is

((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 0))

If the number of leaves of the reduction tree is not a power of two, then the effect is as if the number of identity values for the operation necessary to bring the number of leaves up to a power of two were added on the right (i.e., the rightmost value or expression is associated with a level closer to the root of the tree). Other examples, using shorter and longer sequences, are

((1 + 2) + (3 + 4)) + (5)

((1 + 2) + (3 + 4)) + (5 + 6)

((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))

The identity values are: zero and **false** for **sum**, one and **true** for **product**, positive infinity for **least**, negative infinity for **greatest**, and the empty array or stream for **catenate**.

7.4.1.2.4. ARRAY OF

The **array of** prefix signifies that elements of the sequence described earlier are all to be returned in the form of an array. Each element of the sequence becomes an array element, where the filling begins at index position one and continues sequentially from there. If a masking clause associated with this result form produces an error, the resulting array will be erroneous but all elements of the array computed prior to the first masking error will be visible.

7.4.1.2.5. STREAM OF

The **stream of** prefix signifies that elements of the sequence described earlier are all to be returned in the form of a stream. Each element of the sequence

becomes a stream element, the order of the elements of the stream is the same as the order of values in the sequence used to produce the results. If a masking clause associated with this result form produces an error, the resulting stream will be erroneous but all elements of the stream computed prior to the first masking error will be visible.

7.4.2. The product form of the FOR construct

7.4.2.1. Description

The product form of the **for** construct is a special version of the non-product form that provides a more concise way to specify array and stream index and element sets. The values in the result tuple depend only on the values defined by the in-exp-list and decldef-part.

Syntax:

```
iteration-exp   ::=  for in-exp-list
                     [ decldef-part ]
                     returns return-exp-list
                     end for

in-exp-list     ::=  in-exp
                |    in-exp  dot in-exp [ dot in-exp ] ...
                |    in-exp  cross in-exp [ cross in-exp ] ...

in-exp          ::=  value-name in expression [ at index-list ]

index-list      ::=  value-name [ , value-name ] ...
```

All computations that can be expressed by the product form can also be expressed by the non-product form. The converse is not true.

This construct may introduce zero or more index value names of type **integer**, zero or more element value names of the base type of an array or stream, and zero or more temporary value names, the latter in the same manner as in a **let** block. At least one index or element value name must be introduced. That is, there must be at least one in-exp following the word **for**.

The index, element, and temporary value names must all be different. Their scopes are the entire construct less any inner blocks that reintroduce the same value name.

If any of the in-exp range expressions produces an error (i.e., either of the bounds of an integer range is an error value, the array expression is an error value, or the stream expression is an error value), the results in the return value tuple will be **error** of the appropriate types. However, for erroneous array and stream generators, if the array or stream has accessible or hidden values,

the expression will execute over those values and produce any array and stream results with accessible or hidden values, respectively. If the range is valid but empty (i.e., an integer index range has HI < LO, the array expression in an array range has no elements, or the stream expression has no elements), each type of range result produces the appropriate default value as listed below:

value of		<name>	**error**
	sum	<name>	0 or **false** (depending on type)
	product	<name>	1 or **true** (depending on type)
	least	<name>	**error**
	greatest	<name>	**error**
	catenate	<name>	empty array (low and high bounds 1 and 0, respectively) or empty stream
array of		<name>	empty array, low and high bounds 1 and 0, respectively
stream of		<name>	empty stream

7.4.2.2. IN expressions

Each in-exp makes a set of indices and/or elements of an array or stream available for use in computing a result value. There are three variations on the theme. Any or all variations may be specified in a particular **for** construct.

7.4.2.2.1. Index IN expressions

This introduces one or more index value names of type **integer**.

Syntax:

in-exp ::= value-name **in** expression

The expression appearing after the word **in** must be of arity two. The first and second values are the lower and upper bounds, inclusive, for the index. For each number within those bounds, the index is defined to be that number, the definitions of the temporary names that depend on that index are made, and all the return expressions that depend on that index are evaluated. Each value in the range expression must be of type **integer**.

7.4.2.2.2. Array element IN expressions

This introduces an element value name of the base type of an array.

Syntax:

in-exp ::= value-name **in** expression [**at** index-list]

index-list ::= value-name [, value-name] ...

The element name appears before the word **in**. The expression must be of arity one and of an **array** type. In this case, the shape of the array defines the range of execution of the **for** expression. If the array has one dimension, that dimension defines the range of execution. The body of the **for** executes once for each element of the array and during execution the identifier "value-name" is bound to the corresponding array element. If the optional clause, **at**, is present, the value names following **at** denote index values of type **integer** corresponding to the current element value's position in the array. It is an error if the number of value names in the index list is greater than the number of dimensions of the array expression.

If the array given in the expression is multi-dimensional, and no **at** clause is given, the default range of the **for** expression is over the outermost dimension (that dimension that varies most slowly in a create-by-elements operation) of the array, since it must be defined as an array of arrays. If the **at** option is specified, then the range of the **for** expression is the cross product over the number of ranges specified by the number of names in the index list. For example, assume the following declarations are in effect:

```
Array1 : array [ integer ]
Array2 : array [ array [ array [ integer ] ] ]
```

The following range headers would have the meaning indicated:

1. Elem **in** Array1	=> process all elements of the array
2. Elem **in** Array1 **at** I	=> same as above, but as values are bound to Elem, bind the corresponding array index value to I
3. Elem **in** Array2	=> process across the outermost dimension of Array2
4. Elem **in** Array2 **at** I	=> process the same dimension, but bind to I as described above
5. Elem **in** Array2 **at** I, J	=> process two outer dimensions
6. Elem **in** Array2 **at** I, J, K	=> process all dimensions

Notice that the type of "Elem" varies among these examples. In the first two cases and the last case, Elem's type is **integer**. In the third and fourth cases its type is **array** [**array** [**integer**]], and in the fifth case its type is **array** [**integer**].

7.4.2.2.3. Stream Element IN Expressions

This introduces an element value name of the base type of a stream.

Syntax:

in-exp ::= value-name **in** expression [**at** value-list]

The element name appears before the word **in**. The expression must be of arity one and of a **stream** type. The body of the **for** executes once for each element of the stream and during execution the identifier "value-name" is bound to the corresponding stream element. If the optional clause, **at**, is present, the name following **at** denotes an offset value of type **integer** corresponding to each element value. The first element of the stream is associated with an offset value of one, the second with two, and so on.

The stream range header defines a set of index values, just as the index range header and array range header do. The lower bound of this range is one and the upper bound is defined by stream_size(expression).

7.4.2.2.4. DOT products in IN expression lists

A sequence of index expressions may be given in the in-exp-list, separated by the reserved word **dot**. This range definition produces an inner, or dot, product range of index expressions. For example

```
I in 1,10    dot    J in 11,20
```

defines the set of ten index pairs [1,11], [2,12], [3,13], etc. If all the index ranges do not contain the same number of values, the number of tuples is the number of values in the widest range. Narrower ranges are extended with **error**[**integer**] values to equal the number of values in the widest range. For example:

```
I in 1,6    dot    J in 25,29    dot    K in 4,6
```

defines six triples :

I	J	K
1	25	4
2	28	5
3	27	6
4	28	**error**[**integer**]
5	29	**error**[**integer**]
6	**error**[**integer**]	**error**[**integer**]

Any index expression may be **dot**ted with any other, that is, an integer range may be used with an array range and a stream range, etc. However, **dot** and **cross** products cannot be intermixed in one in-exp-list. Note that use of the "**at**" clause with multiple indices is considered to be an implied **cross** product and therefore cannot be used in conjunction with the **dot** option. For example, the following **for** header is illegal.

```
for    Aelem in Aarray at I , J    dot
       Belem in Barray at K , L

       .... ( ILLEGAL SISAL ! )
```

Such an expression would have to be written so as to compute the values of I and L given values for I and J.

Name scoping is influenced by the use of the **dot** notation. Each index expression introduces a new identifier name that is local to the entire **for** expression. However, any such names introduced cannot be used later in the same header. This rule insures that all index ranges defined by **dot** are independent of each other.

7.4.2.2.5. CROSS products in IN expression lists

A sequence of index expressions may be given in the in-exp-list separated by the reserved word **cross**. The index range thus formed is the Cartesian, or outer, product of the index expressions. For example

I **in** 1,10 **cross** J **in** 11,20

defines an index range consisting of the tuples [I,J], where 1<=I<=10 and 11<=J<=20.

A **in** Array1 **cross** B **in** Array2

defines an index range consisting of the tuples [I,J], where

array_liml(Array1) <= I <= array_limh(Array1) and
array_liml(Array2) <= J <= array_limh(Array2).

A form which allows explicit use of the index values is

A **in** Array1 **at** I **cross** B **in** Array2 **at** J.

Similar forms may be used to cross stream ranges. Any index expression may be **crossed** with any other, that is, an integer range may be used with an array range and a stream range, etc. However, it is illegal to use both **dot** and **cross** in the same in-exp-list.

Name scoping is influenced by the use of the **cross** notation. Each index expression introduces a new identifier name that is local to the entire **for** expression. Unlike the **dot** case, these identifiers are immediately available for use in defining later ranges within the **for** header. So for example, the following header would allow processing of a stream of arrays.

```
for   Selem in StreamS   cross
      Aelem in Selem

      ....
```

7.4.2.3. Result Values

The result value of the product form of the **for** construct is the tuple of values defined by the return-exp-list, as for the non-product form. The **value of**, **array of**, and **stream of** prefix expressions behave in a manner similar to when they

appear in the non-product form, however, the range expression headers and **cross** product options require more detailed explanations. In the non-product form, the **for** expression execution defines a natural ordering on the sequence of values that can be used to produce results. In the product form, since all elements of the range are independent of each other, the natural ordering is not so clear. However, to insure determinate behavior, SISAL does impose an ordering of the results produced by the product form.

The basic case is the single range expression. In this case, each result clause is evaluated once for each element of the range. The results of these evaluations are placed in a sequence, using the range expression's ordering (from least to greatest) as the sequence ordering. As in the non-product form, masking clauses may be used to eliminate specific results from this sequence. Therefore the resulting sequence may be shorter than the original range size. SISAL also allows the **old** option on results; it has the same influence on the result sequence as it had in the non-product case. (This decision is mostly for completeness, we do not expect this particular combination of features to be particularly useful.) **Dot** range expressions produce results in an analogous manner.

Cross range expressions differ in that they produce a sequence of sequences in their pattern for generating results. If there are no modifying clauses, the structure of the results exactly matches the structure of the **cross** ranges. The outermost sequence corresponds to the first range expression in the header. Masking clauses may be applied as in the earlier cases, but the results may be somewhat surprising. If a result element is masked, it causes a compression of valid elements in the last dimension of the ranges described in the header. Compressions can only occur in this dimension.

Given these basic extensions for defining the ordering on results produced by the product form of the **for** expression, each of the specific types of result clauses operates essentially the same as in the non-product case, except as described below.

7.4.2.3.1. VALUE OF

If **value of** is not followed by one of the reduction operators, the result produced by this clause is the last element of the sequence as defined above. In the case of **cross** product ranges, it is the value of the last element of the last sequence.

If **value of** is followed by one of the reduction operators, the operation is performed on the collection of values in the sequence. The order in which the reduction operation is performed is subject to the **left**, **right**, and **tree** constraints, if specified. In the case of **cross** product ranges, the reduction operation is first performed on all elements within the lowest dimensioned sequences and then successively applied to all higher level dimensioned sequences.

7.4.2.3.2. ARRAY OF

Array of produces an array having size equal to that of the sequence of results as described above and containing exactly those values in the order defined by the sequence. The lower bound of the array is equal to the index value from the range clause that produced the first element of the sequence. If

the **for** defines a **cross** product range, then the resulting array is multi-dimensional, the shape corresponding to the structure of the sequence of sequences defined above.

7.4.2.3.3. STREAM OF

Stream of produces a stream having size equal to that of the sequence of results as described above and containing exactly those values in the order defined by the sequence. If the **for** defines a **cross** product range, then the resulting stream is multi-dimensional, the shape corresponding to the structure of the sequence of sequences defined above.

8. FUNCTION DEFINITIONS AND COMPILATION UNITS

A SISAL program consists of a collection of type and function definitions. For compilation convenience, a program can be distributed among several compilation units, each of which contains type and function definitions. A compilation unit explicitly defines which of the functions appearing in its scope are to be accessible to other units (via **defines**) and which functions may be used within the unit without definition (**globals**). Compilation units and function definitions may also include **forward function** declarations to allow convenient handling of recursive and mutually recursive functions within a compilation unit or function definition. The syntax of a compilation unit is as follows.

compilation-unit ::= **define** function-name-list
[type-def-part]
[**global** function-header] ...
function-def ...

function-name-list ::= function-name [, function-name] ...

function-def ::= **forward function** function header
| **function** function-header
[type-def-part]
[function-def] ...
expression
end function

type-def-part ::= type-def [; type-def] ... [;]

type-def ::= **type** type-name = type-spec

function-header ::= function-name ([decl-list] **returns** type-list)

decl-list ::= decl [; decl] ... [;]

type-list ::= type-spec [, type-spec] ...

function-name ::= name

Example:

```
define    Merge, Push, Pop, Empty
type      Stack = array [ integer ]
global    Find ( Object : Stack; Data : integer returns boolean )
forward   function Empty ( Object : Stack returns boolean )
```

```
function    Push ( Data : integer; Object : Stack returns Stack )
        array_addh ( Object, Data )
end function

function Pop ( Object : Stack returns integer, Stack )
        if Empty(Object)
        then
                error[integer], Object
        else
                Object[ array_limh(Object) ], array_remh(Object)
        end if
end function

function Empty ( Object   : Stack returns boolean )
        array_size(Object) = 0
end function

function Merge ( Stk1, Stk2 : Stack returns Stack )
        for
                NewStack := Stk1 ;
                SrcStack := Stk2 ;
        until  Empty ( SrcStack )
        repeat
                Elem, SrcStack  :=  Pop ( old SrcStack ) ;
                NewStack := if  Find (Elem, old NewStack)
                                then  old Newstack
                                else  Push ( Elem, old Newstack )
                                end if ;
        returns
                value of  NewStack
        end for
end function
```

In this example, the compilation unit defines four functions usable by other compilation units and has no purely local functions. This unit must uses one function (Find) in Merge that it does not define, hence the use of the **global** denotation. The forward declaration allows the Pop function to call Empty without prior definition of the entire body. Note that by reordering in this case the need for the **forward function** can be eliminated. Also, notice that because SISAL uses structural type checking, there is no need to export type definitions.

8.1. The header and value transmission

The list of formal arguments and their type specifications appear in the header between the left parenthesis and the word **returns**. These declarations are separated from each other by semicolons. Each declaration may contain several value names, which are separated from each other by commas.

The scope of the formal arguments is the body of the function (the expression), less any inner constructs which reintroduce the same value name. Their types are as given in the header declarations, and their values are the values of the arguments given at function invocation. The types of the returned values are given in the list of type specifications, separated by commas, appearing after the word **returns**. This list of types must conform to the body. In every

invocation of a function, the number and types of the arguments and returned values must match those of the definition.

The meaning of a function invocation is as follows: If the function F is defined by

```
function F (A1 : T1; ...; An : Tn returns S1, ... , Sk )
   body-expression
end function
```

then, assuming the definition is correct and conforms to its invocation, the invocation

```
F(argument-expression)
```

is equivalent to

```
let
   A : T1, ..., An : Tn := argument-expression
in
   body-expression
end let
```

8.2. The GLOBAL declaration

All functions used in a unit that are not defined in that unit must be declared in a global declaration. This declaration consists of the word **global** followed by a copy of the function's header, which is used by the translator for type checking. **Global** may be used only in the context of a compilation unit. It is legal to identify a function as being **global**, and then within the compilation unit define a function of the same name. In this case, the local definition overrides the **global** definition. This choice allows a compilation unit to **define** a **global** function (a situation that would arise if all units use a macro facility to include a file containing all globally defined functions).

Example:

```
define Tan
global Cos (Q : real returns real)
global Sin (Q : real returns real)

function Tan (X : real returns real)
      Sin(X) / Cos(X)
end function
```

This compilation unit defines the function Tan. Since it uses the functions Sin and Cos, which are not defined here, they must appear in global declarations (they must be defined in other compilation units or accessed in a subroutine library). The global declarations contain the headers for Sin and Cos, just as

they might appear in the definitions of those two functions. The formal arguments appearing in the headers ("Q" in the preceding example) have no significance; they are included only for syntactic consistency. The intention is that the headers be copied verbatim from the units defining Sin and Cos into the unit defining Tan.

All functions used anywhere within a compilation unit that are not defined in that unit must appear in a **global** clause at the beginning of the compilation unit.

8.3. The FORWARD FUNCTION declaration

The effective use of mutual recursion (see Section 8.5) requires that the name and argument and result types of a function be available before the actual definition of that function. The **forward function** declaration provides that information. This declaration permits type checking to proceed when a call is made to a function whose body has not yet been seen.

Example:

```
forward function X ( A : real returns real )

function Y ( B : real returns real )
  X(B)
end function

function X ( A : real returns real )
  Y(A)
end function
```

In the above case the recursion is infinite. A **function** definition corresponding to the **forward function** declaration must appear before the end of the enclosing function or compilation unit. The number, names, and types of the arguments and the types of the results provided in the **forward function** declaration and its corresponding **function** definition must match, in the order in which they were declared.

8.4. Inheritance of data, type definitions, and global and forward declarations

A function has access only to the data presented to it in its invocation. No data values are imported from any enclosing function definition.

Type declarations and definitions made at the outermost level of a compilation unit or in a function definition are inherited by all subsidiary functions. A redeclaration or redefinition in a nested function of a type name already declared or defined in an outer context is not permitted.

Import declarations made in a compilation unit are inherited by all function definitions within that compilation unit.

Forward declarations made in any function or compilation unit are inherited by all nested functions. Forward declarations made at the outermost level of a compilation unit are inherited by all functions subsidiary to them. A

redeclaration in an internal function of a forward declared function name already defined in an outer scope is not permitted.

Function names defined in any function or compilation unit are inherited by all nested functions. All function names at the outermost level of a compilation unit must be distinct. A redefinition of a function already defined in an outer scope is not permitted.

8.5. Scope of function definitions

The scope of a function definition identifies the parts of the compilation unit that can invoke that function. The scope of a function definition in SISAL includes the function's own body, all functions declared after that function at the same declaration level, and the body of the immediately enclosing function. The scope of a function declared in a **forward function** declaration includes (in addition) all functions declared after the forward declaration. Note that this permits recursion and mutual recursion.

Example:

```
type MT1 = <type-spec>

global EF1 ( <EF1-header> )

global EF2 ( <EF2-header> )

forward function M2 ( <M2-header> )

function M1 ( <M1-header> )

  type T = <type-spec>

  function G ( <G-header> )
      type U = <type-spec>

      function M ( <M-header> )
        function N ( <N-header> )
        <N-body>
        end function

      <M-body>
      end function

 <G-body>
 end function

 function H ( <H-header> )
     function P ( <P-header> )
     <P-body>
     end function

 <H-body>
 end function
```

<M1-body>
end function

function M2 (<M2-header>)
<M2-body>
end function

The legal function invocations are as follows.

the body of	may invoke functions
M1	M1, M2, EF1, EF2, G, H
G	M1, M2, EF1, EF2, G, M
M	M1, M2, EF1, EF2, G, M, N
N	M1, M2, EF1, EF2, G, M, N
H	M1, M2, EF1, EF2, G, H, P
P	M1, M2, EF1, EF2, G, H, P

This example illustrates the possible non-symmetry of access between various function bodies. Since H is not declared to be **forward**, G (and all internal scopes to G) cannot call H, even though they are at the same lexical level and in the same definition block. H can, however, call G because that definition is specified before H. Since P is internal to H, P can access any object that H can.

The legal uses of defined types are as follows.

the header of	may use defined types
M1	MT1,
G	MT1, T
M	MT1, T, U
N	MT1, T, U
H	MT1, T
P	MT1, T

the body of	may use defined types
M1	MT1, T
G	MT1, T, U
M	MT1, T, U
N	MT1, T, U
H	MT1, T
P	MT1, T

The compilation units comprising a program are translated separately. The manner in which functions are linked into a complete program is implementation dependent.

redeclaration in an internal function of a forward declared function name already defined in an outer scope is not permitted.

Function names defined in any function or compilation unit are inherited by all nested functions. All function names at the outermost level of a compilation unit must be distinct. A redefinition of a function already defined in an outer scope is not permitted.

8.5. Scope of function definitions

The scope of a function definition identifies the parts of the compilation unit that can invoke that function. The scope of a function definition in SISAL includes the function's own body, all functions declared after that function at the same declaration level, and the body of the immediately enclosing function. The scope of a function declared in a **forward function** declaration includes (in addition) all functions declared after the forward declaration. Note that this permits recursion and mutual recursion.

Example:

```
type MT1 = <type-spec>

global EF1 ( <EF1-header> )

global EF2 ( <EF2-header> )

forward function M2 ( <M2-header> )

function M1 ( <M1-header> )

   type T = <type-spec>

   function G ( <G-header> )
        type U = <type-spec>

        function M ( <M-header> )
          function N ( <N-header> )
          <N-body>
          end function

        <M-body>
        end function

   <G-body>
   end function

   function H ( <H-header> )
        function P ( <P-header> )
        <P-body>
        end function

   <H-body>
   end function
```

<M1-body>
end function

function M2 (<M2-header>)
<M2-body>
end function

The legal function invocations are as follows.

the body of	may invoke functions
M1	M1, M2, EF1, EF2, G, H
G	M1, M2, EF1, EF2, G, M
M	M1, M2, EF1, EF2, G, M, N
N	M1, M2, EF1, EF2, G, M, N
H	M1, M2, EF1, EF2, G, H, P
P	M1, M2, EF1, EF2, G, H, P

This example illustrates the possible non-symmetry of access between various function bodies. Since H is not declared to be **forward**, G (and all internal scopes to G) cannot call H, even though they are at the same lexical level and in the same definition block. H can, however, call G because that definition is specified before H. Since P is internal to H, P can access any object that H can.

The legal uses of defined types are as follows.

the header of	may use defined types
M1	MT1,
G	MT1, T
M	MT1, T, U
N	MT1, T, U
H	MT1, T
P	MT1, T

the body of	may use defined types
M1	MT1, T
G	MT1, T, U
M	MT1, T, U
N	MT1, T, U
H	MT1, T
P	MT1, T

The compilation units comprising a program are translated separately. The manner in which functions are linked into a complete program is implementation dependent.

APPENDIX A – SISAL Syntax

```
compilation-unit    ::=  define function-name-list
                         [ type-def-part ]
                         [ global function-header ] ...
                         function-def ...

function-name-list  ::=  function-name [ , function-name ] ...

function-def        ::=  forward function function-header
                    |    function function-header
                         [ type-def-part ]
                         [ function-def ] ...
                         expression
                         end function

type-def-part       ::=  type-def [ ; type-def ] ... [ ; ]

type-def            ::=  type type-name = type-spec

function-header     ::=  function-name ( [ decl-list ] returns type-list )

decl-list           ::=  decl [ ; decl ] ... [ ; ]

type-list           ::=  type-spec [ , type-spec ] ...

type-spec           ::=  basic-type-spec
                    |    compound-type-spec
                    |    type-name

basic-type-spec     ::=  boolean
                    |    character
                    |    double_real
                    |    integer
                    |    null
                    |    real

compound-type-spec  ::=  array [ type-spec ]
                    |    stream [ type-spec ]
                    |    record [ field-spec [ ; field-spec ] ... [ ; ] ]
                    |    union [ tag-spec [ ; tag-spec ] ... [ ; ] ]

field-spec          ::=  field-name [ , field-name ] ... : type-spec
```

```
tag-spec            ::=  tag-name [ , tag-name ] ... [ : type-spec ]

expression          ::=  simple-expression [ , simple-expression ] ...

simple-expression   ::=  primary [ binary-op primary ] ...

unary-op            ::=  + | - | ~

binary-op           ::=  < | <= | > | >= | = | ~=
                     |   + | - | ||
                     |   * | / | &
                     |   ||

primary             ::=  constant                    (arity 1)
                     |   value-name                  (arity 1)
                     |   ( expression )              (arity of expression
                     |                                in parentheses)
                     |   invocation                  (arity is the number of
                     |                                values returned)
                     |   array-ref                    (arity 1)
                     |   array-generator              (arity 1)
                     |   stream-generator             (arity 1)
                     |   record-ref                   (arity 1)
                     |   record-generator             (arity 1)
                     |   union-test                   (arity 1)
                     |   union-generator              (arity 1)
                     |   error-test                   (arity 1)
                     |   prefix-operation            (arity 1)
                     |   conditional-exp
                     |   let-in-exp
                     |   tagcase-exp
                     |   iteration-exp
                     |   old value-name              (arity 1)
                     |   unary-op primary            (arity 1)

invocation          ::=  function-name ( [ expression ] )

array-ref           ::=  primary [ expression ]

array-generator     ::=  array type-name [ ]
                     |   array [ type-name ] [ expr-pair ]
                     |   primary [ expr-pair [ ; expr-pair ] ... [ ; ] ]

expr-pair           ::=  expression : expression
```

```
stream-generator  ::=  stream type-name [ ]
                  |    stream [ type-name ] [ expression ]

record-ref        ::=  primary . field-name

record-generator  ::=  record [ type-name ] [ field-def [ ; field-def ] ... [ ; ] ]
                  |    primary replace [ field : expression
                                         [ ; field : expression ] ... [ ; ] ]

field-def         ::=  field-name : expression

field             ::=  field-name [ . field-name ] ...

union-test        ::=  is tag-name ( expression )

union-generator   ::=  union type-name [ tag-name [ : expression ] ]

error-test        ::=  is error ( expression )

prefix-operation  ::=  prefix-name ( expression )

let-in-exp        ::=  let
                       decldef-part
                       in
                       expression
                       end let

decldef-part      ::=  decldef [ ; decldef ] ... [ ; ]

decldef           ::=  decl
                  |    def
                  |    decl [ , decl ] ... := expression

decl              ::=  value-name [ , value-name ] ... : type-spec

def               ::=  value-name [ , value-name ] ... := expression

tagcase-exp       ::=  tagcase [ value-name := ] expression
                       tag-list : expression
                       [ tag-list : expression ] ...
                       [ otherwise : expression ]
                       end tagcase

tag-list          ::=  tag tag-name [ , tag-name ] ...
```

```
conditional-exp ::= if expression then expression
                    [ elseif expression then expression ] ...
                    else expression
                    end if

iteration-exp   ::= for initial                    (non-product form)
                    decldef-part
                    iterator-terminator
                    returns return-exp-list
                    end for

                |   for in-exp-list                (product form)
                    [ decldef-part ]
                    returns return-exp-list
                    end for

iterator-terminator ::= iterator termination-test
                    |   termination-test iterator

iterator        ::= repeat
                    iterator-body

termination-test ::= while expression
                 |   until expression

iterator-body   ::= decldef-part

in-exp-list     ::= in-exp
                |   in-exp dot in-exp [ dot in-exp ] ...
                |   in-exp cross in-exp [ cross in-exp ] ...

in-exp          ::= value-name in expression [ at index-list ]

index-list      ::= value-name [ , value-name ] ...

return-exp-list ::= return-clause ...

return-clause   ::= [ old ] return-exp [ masking-clause ]

masking-clause  ::= unless expression
                |   when expression

return-exp      ::= value of [ [ direction ] reduction-op ] expression
                |   array of expression
                |   stream of expression
```

```
direction       ::=  left
                |    right
                |    tree

reduction-op    ::=  sum
                |    product
                |    least
                |    greatest
                |    catenate

constant        ::=  false
                |    nil
                |    true
                |    integer-number
                |    real-number
                |    character-constant
                |    character-string-constant
                |    error [ type-spec ]

prefix-name     ::=  character          (arity 1)
                |    double_real       (arity 1)
                |    integer            (arity 1)
                |    real               (arity 1)

function-name   ::=  name

field-name      ::=  name

tag-name        ::=  name

type-name       ::=  name

value-name      ::=  name
```

APPENDIX B – Implementation Limits

1. Streams

An implementation may choose to restrict the base type of a stream type to those whose run time bit size may be computed at translation time. This means that the base type of a stream may not be

1. an **array** or **stream** type
2. a type that contains **array** or **stream** types as part of its definition, e.g., a **record** containing an **array** or **stream** component.

The implications of this for other parts of the language are as follows.

1. The **stream of** return expression prefix may not be used with a **cross** product in-exp-list, since the result is a stream of streams. See Sections 7.4.1.2.5 and 7.4.2.3.3.

2. The **stream of** return expression prefix may not be used with an expression which does not meet the restrictions on the base type of a stream given above.

An implementation may enforce the above restriction for streams used only in certain implementation-defined ways. For example, it may enforce the restriction only for streams used for Input/Output purposes.

APPENDIX C – Pragmas for Various Implementations

A pragma (that is, a compiler directive) may be specified within a comment by including a dollar sign character '$' immediately after the percent '%' character, followed by the pragma.

1. VAX-11 SISAL

The syntax of a pragma is identical to that of a list of function invocations, separated by commas. The pragma names replace the function names.

```
comment-pragma    ::=   %$ pragma-list
pragma-list       ::=   pragma [ , pragma ] ...
pragma            ::=   pragma-name ( expression )
pragma-name       ::=   name
```

Any text following the last pragma in the list is treated as a genuine comment (i.e., ignored). Any error encountered during interpretation of the pragma-list will cause the compiler to produce a warning and ignore the offending pragma.

The following pragmas are supported.

1.1. INCLUDE(string)

This pragma may appear anywhere in the compilation unit. It allows access to the text of a SISAL source file during the translation of another. The pragma is useful when the same information is used by several compilation units. The contents of the INCLUDEd file are inserted at the point where the translator encounters the INCLUDE pragma.

The character string constant denoted by the string parameter must be a valid file specification for the system running the translator.

When the translator finds the INCLUDE pragma, it stops reading from the current file and begins reading from the INCLUDEd file. When the translator reaches the end of the INCLUDEd file, it resumes translation at the point in the original file following the comment containing the INCLUDE pragma.

Example

```
%$ INCLUDE( "SIS$LIBRARY:SISLIB.ENV" )
```

The contents of the file specified by the VAX/VMS file specification SIS$LIBRARY:SISLIB.ENV are inserted in the SISAL source file.

1.2. SUBRANGE(low-bound, high-bound)

This pragma has meaning only when it occurs before a type specification that resolves to **integer** or a subrange thereof. It informs the compiler that values of that type may not be outside the range defined by low-bound and high-bound, inclusive. Both low-bound and high-bound must be integer-numbers as defined in Section 3. If low- bound is greater than high-bound, a warning is generated and the pragma ignored.

Example

```
type  Bit3   = %$ SUBRANGE(-4, 3)
               integer
```

defines an **integer** type whose values are in the range -4 to 3, inclusive.

1.3. PACKED()

This pragma has meaning only when it occurs before a **record** type specification. It instructs the compiler to pack the components of the record into as few bits as possible. The number of bits used for each component and their alignment depends on the target machine.

Example assuming the previous definition of the subrange type Bit3

```
type  Packed_Rec  = %$ PACKED()
                    record[ I : Bit3; J : Bit3 ]
```

The minimum size of this record is six bits.

1.4. MAIN()

This pragma has meaning only when it occurs before a name in a **define** list. It informs the compiler that the named function represents the entry point of the computation, that is, it is the function that is executed by the surrounding environment.

Example

```
define      Func_A,
            %$ MAIN()
                  Func_Main,
                  Func_B
```

The function Func_Main will be invoked first by the surrounding environment.