SISAL Reference Manual

Language Version 2.0

A. P. W. Böhm
R. R. Oldehoeft
Computer Science Department
Colorado State University

D. C. Cann
J. T. Feo
Computer Research Group
Lawrence Livermore
National Laboratory

Notice

This report was partially supported by The Office of Energy Research (United States Department of Energy) under contract No. W-7405-Eng-48 to Lawrence Livermore National Laboratory. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This work was partially supported by U. S. Army Research Office contract DAAL03-86-K-0101 at Colorado State University. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Contents

Chapter 1

Introduction

In this report we describe the SISAL 2.0 programming language. The project began in 1982 as a cooperative venture by Digital Equipment Corporation, the University of Manchester (England), Lawrence Livermore National Laboratory, and Colorado State University. The project's goals evolved to become:

- Define a general-purpose functional language [11] that can run efficiently on conventional and novel parallel architectures;
- Define a dataflow graph intermediate form independent of language and target architecture [17, 20];
- Develop optimization techniques for high-performance applicative computing [4, 5, 15, 18, 19];
- Develop a thread management environment to support dataflow-style parallel computing on conventional shared-memory multiprocessors [13, 16];
- Achieve sequential and parallel execution performance competitive with programs written in conventional languages [3, 5];
- Validate the functional style of programming for large-scale scientific applications [6].

The last three goals set SISAL apart from other efforts. We have shown that the dataflow approach can profitably be applied outside the classic dataflow world, without the use of specialized hardware. SISAL runs on uniprocessors, shared-memory multiprocessors [13], the Cray-X/MP [2, 10], the Warp systolic processor [8] as well as Manchester's dataflow machine [9]. Development continues for the Connection Machine [7] and some experimental systems [1].

High performance is possible without resorting to imperative programming tricks that make existing programs impossible to parallelize effectively. While functional language features contribute to the "truth and beauty" of programs, performance has been a problem because naive implementations exacerbate a new set of potential inefficiencies. The performance of our optimizers results in SISAL programs that regularly outperform their counterparts written in C and compete well with those written in FORTRAN. Parallel execution is highly efficient as long as the underlying algorithm has exploitable parallelism.

It is now time to define SISAL 2.0. In working with SISAL 1.2 since 1985, we have identified language features whose redefinition will allow for easier expression or better potential for high performance execution. We have included useful features available in other functional languages for improved expressibility without diminishing performance. Finally, the inclusion of non-SISAL modules allows access to extant libraries and non-functional programming paradigms when appropriate [14].

This chapter introduces the importance of the functional approach to parallel computing, gives an overview of the new SISAL, and identifies contributors and major contacts in the ongoing SISAL project. Most of the rest of the report describes the various elements of the revised SISAL language, and exemplifies use of these features. Appendices provide a syntactic summary, give a set of implementation notes, supply a set of longer sample programs, and describe the FIBRE format for values to be read by or emitted from SISAL programs.

The language definition in this report will surely change. As implementation proceeds, we will encounter difficulties that require syntactic or semantic changes. As we and others see new ways to express useful concepts, the language definition will need to accommodate them. Other researchers will point out needed changes, and we will respond. However, we believe that this report describes a language that is nearly complete and correct.

1.1 Functional Languages and Parallel Computing

A functional programming language enables the description of a computation to be written entirely with expressions. A programmer uses built-in operations along with additional explicit function definitions to describe a mathematical computation using input values to define intermediate values and to produce result values. The notion of *referential transparency* is the important characteristic distinguishing the functional approach from the familiar imperative computing paradigm. These two expressions:

$$f(x,y,z) \ + \ f(x,y,z)$$

$$\textbf{let}\ a \ := \ f(x,y,z)\ \textbf{in}\ a \ + \ a\ \textbf{end let}$$

have the same value because a and $f(x,y,z)$ may substitute for each other freely.

Some related notions arise from this. Functions produce values but introduce *no side effects*. We may abbreviate an expression by associating a name to be used in its place (as above), but the name retains this value throughout its existence—this is *single assignment*. Such a name is bound to a value, not a memory location, so there is *no aliasing*, that is, only a single name for each value.

In the imperative paradigm, values are stored in memory locations, and computation proceeds by altering these values. Language constructs define a single sequence for altering this implicit state, so a determinate program results. In the functional approach, state is explicit in the expressions and names of values. Recursion provides a mechanism for establishing a new explicit state context with different values; carefully designed iterative expressions can have the same effect in a more convenient way.

Since functions are an important element, we want to manipulate them as we do other values; *higher-order functions* take functions as parameters and yield functions as results. In addition to defining them, evaluating them, and passing them as parameters to other functions, functional languages allow expressions to be *function-valued*. Hence functions may produce other functions for later use.

Another common feature of functional languages is *type inference*. Instead of requiring the programmer to provide the type of each name explicitly, it is possible to infer the types of some values from the types of input values and the transformations done by the operations. In a closely related way, functions can be written using *typesets*, lists of alternative type specifications for formal parameters. At each function invocation, a concrete type is deduced from each actual parameter and substituted in the definition if a match occurs with a typeset element. Type inference then makes all the types of all values in the function known, and compilation of a version of the function for this invocation can go forward.

Proponents of the functional paradigm believe that these language features yield important benefits:

- Programs are more concise. This is important because smaller programs are easier to write and maintain than larger ones. The expressive power associated with higher-order functions and typesets with inference is the main reason. But one must take care to avoid an oppressively terse and arbitrary syntax that can result in a loss in comprehensibility.

- Programs may be reasoned about, because function definitions are mathematically sound, and side effects and aliasing do not get in the way. The long-sought goal of proving nontrivial programs to be correct is achievable with functional languages.

- Programs may be automatically parallelized to exploit multiple processor machines. The lack of an imperative sequencing order leaves only data dependency as the mechanism defining the order of operations. Since side effect and aliasing problems are banished by definition, no complex analysis is required to uncover potential parallelism. Data dependency ordering includes elementary operations, so the finest granularity of parallelism is easily found and can be exploited in a dataflow system. Alternatively, coarser parallelism granularity may be exploited by choosing other packages for the sequential threads composing the program in execution.

We have focused primarily on the last benefit, while other research groups have concentrated on the second benefit. We expect that the first benefit will be exploitable in the revised SISAL described in this report.

1.2 SISAL Language Overview

We describe the general features of SISAL here. Specific syntax and examples are found in the various chapters that follow.

1.2.1 Program structure

An executable SISAL entity comprises one or more separately compilable *compilation units*, which include a program, interfaces and modules. The simplest is a *program*: a set of declarations that define entities imported from other modules, some types, and some functions. Any function in a program may be the starting point for execution. At this outermost level, function parameters are values obtained from the operating system level, and function results are produced at that level.

A *module* is syntactically similar to a program, but is not a starting point for execution. It pairs with an *interface* to export some of its type and function names. The associated interface has the same name as the module, and public types are re-defined there. Public functions have their headers repeated in the interface. An interface may also specify that the public functions of its module will be accessed by software not written in SISAL. This allows a software system to have subsidiary parts written in SISAL.

A stand-alone interface declares the relationship between SISAL and a set of subsidiary code written in other languages. This allows SISAL software to access libraries of already written code.

1.2.2 Types

Data types include the usual scalars (boolean, character, integer, real, double, complex, and double_complex), structured types (records and unions, arrays, and streams), and functions. Structured types may have values of any type as constituents; records and unions have heterogeneous constituents and arrays and streams have homogeneous constituents.

An interface may specify the name of a record or union type for public use, but may prevent the publication of the constituents.

A type may be declared to be one of a set of alternatives. This is useful for writing functions with formal parameter types not given concretely. A function reference will supply known actual parameter types, which are used to complete the compilation of this instance of the function. This facility for *typesets* can simplify production of code that operates on different arithmetic types, for example.

Function values may be parameters to functions and the results of expression evaluation, so function types may be declared by giving the types of all parameters and results.

Several functional programming languages use a complete type inference system wherein the types of all values are inferred from their contexts. The foundation for such systems is given in [12]. After examining these languages, we conclude that the ability to do complete type inference is not compelling enough for us to adopt it. Instead, we require that types of formal parameters and of results for functions be explicitly supplied, and require the component types of structured values, for several reasons. First, explicit typing provides documentation valuable to both program writers and readers. Second, much of the advantage of type inference is available via our explicit typesets. Third, functions which are possible starting points for execution have concrete types, and they form roots from which intra-function type inference and inter-function inference for typesets may proceed. Requiring just this much type information makes complete compile-time typing possible for all SISAL programs. Fourth, type inference in the presence of overloaded operators (in SISAL, for example, + takes integer, real, double, complex, and double_complex scalar operands, as well as conformable arrays and streams of these scalars) is not a well-developed procedure.

1.2.3 Functions

A function is declared by listing its name, the names and types of its formal parameters, and types of its result values. The content of a function is one or more expressions (a *multi-expression*) whose types correspond to the result types. Values are available to the expressions via the formal parameters, not through globally accessed names.

Higher-order function operations are part of SISAL. Functions can be passed to and returned from functions, and be the values of expressions.

1.2.4 Expressions

Expressions are, of course, the heart of the language. Syntax is designed to be as familiar as possible.

Simple Expressions and Name Scoping

Conventional infix operators combine scalar arithmetic values. SISAL supports some type promotion automatically, and provides some predefined type conversion functions.

One can assign the value of any expression to a name, and use the name as shorthand for the expression throughout the scope of the definition. This scoping is done with the **let** construct.

Arrays

SISAL has comprehensive facilities for defining and manipulating array values. An array generator allows the definition of a multidimensional object whose parts form a "tiling" of the overall structure. Arbitrary subarray selection is provided beyond the rectangular subsets available in some other notations. Many infix operators operate element-by-element on array operands, and a useful set of functions on arrays is defined. A subarray update facility allows safe alteration of array values. Many applications are expressible succinctly with these features. Array generation, selection, and update may use *vector subscripts* to refer to arbitrary, non-geometric sections of arrays.

Streams

A stream is a sequence of values produced in order by one expression evaluation, and consumed in the same order by one or more other expression evaluations. Producers and consumers are usually **for** expressions, but short forms for simple streams are also available. To expose the pipelined parallelism that streams make possible, they must be implemented non-strictly. That is, consumer expressions must be started whether or not the producer expression has finished.

1.2.5 Control by Selection

Two constructs for selection are provided in SISAL: **if** and **case** expressions. The arms of an if expression are guarded by boolean expressions, while the arms of a case expression are guarded by the values of the selecting expression. The arms of a single **if** or **case** expression must agree in arity and type, unless the selection is being used for type inquiry.

1.2.6 Control by Distribution and Iteration

A single **for** construct has two forms for potentially parallel, as well as sequential, evaluation. In the first form values are distributed to the bodies of the construct and each body defines values to contribute to the overall result. The second form has dependencies between values defined in one body and used in the successor body. In either form the values from the bodies are collectable into an array or a stream, or reducible to a single value. Array construction in **for** expressions allows permutation of the individual body values.

1.2.7 Errors

SISAL includes standard error processing semantics for managing erroneous computations. However, a SISAL implementation may elect to stop execution when an error is encountered. Each SISAL type has a distinguished value, **error**. Any failed expression evaluation results in **error** of the appropriate type. Error values propagate in a well-defined way when they are operands in computations. Error values can be tested for, and even explicitly assigned to signify other anomalous conditions.

Chapter 2 supplies syntactic notation and introduces lexical definitions. Scalar types and values along with elementary constructs are found in Chapter 3. Chapters 4 and 5 cover arrays and streams, respectively. The versions of **for** expressions are explained in Chapter 6. Chapter 7 covers records and unions, and selection constructs are found in Chapter 8. Functions and related ideas are explained in Chapter 9. Program structuring forms are in Chapter 10. Concepts relating to error values are in Chapter 11. Finally, some ideas for some interesting language features we have not included in this version of SISAL are discussed in Chapter 12.

1.3 Acknowledgements

In addition to the authors, many people have contributed to the design and implementation of SISAL. Researchers world-wide have used SISAL software in their own work. Here we would like to thank the members, past and present, of our organizations.

University of Manchester: John R. W. Glauert, John Gurd, Chris Kirkham, Michael O'Boyle, John Sargeant, Heidi W. H. Tang, Y. M. Teo, Ian Watson.

Lawrence Livermore National Laboratory: Tom DeBoni, Ching-Cheng Lee, James R. McGraw, Pat Miller, John E. Ranelletti, Rea Simpson, Stephen K. Skedzielewski, Michael L. Welcome, R. Kim Yates, Jeremy Uejio, David Zimmerman.

Colorado State University: Stephen J. Allan, Bruce Bigler, Steve Dominic, Dale Grit, Tom Hanson, Seetharaman Harikrishnan, John Lang, Tamara R. Richert.

Thanks also to Andrew Wendelborn from the University of Adelaide for his careful analysis of SISAL features during his visit at Colorado State University, and to Robert Keller for his comments during his time at Lawrence Livermore National Laboratory.

1.4 Addresses for SISAL Contacts

We are available for contact at these addresses:

A. P. W. Böhm Rodney R. Oldehoeft	David C. Cann John T. Feo
Computer Science Department Colorado State University Fort Collins, CO 80523	Computer Research Group L-306 Lawrence Livermore National Laboratory Livermore, CA 94550
{bohm,rro}@CS.ColoState.EDU	{cann,feo}@sisal.LLNL.GOV
303/491-5792	415/422-6389

1.5 References

[1] D. Abramson and G. K. Egan. An overview of the RMIT/CSIRO parallel system architecture. *The Australian Computer Journal*, 20(3), August 1988.

[2] D. C. Cann. Retire Fortran? a debate rekindled. Technical Report UCRL-JC-107018, Lawrence Livermore National Laboratory, April 1991.

[3] D. C. Cann. Vectorization of an applicative language: Current results and future directions. In *Compcon 91*, pages 396–402, February 1991.

[4] D. C. Cann and R. R. Oldehoeft. Reference count and copy elimination for parallel applicative computing. Technical Report CS-88-129, Computer Science Department, Colorado State University, Fort Collins, CO, November 1988.

[5] D. C. Cann and R. R. Oldehoeft. High performance parallel applicative computing. Technical Report CS-89-104, Computer Science Department, Colorado State University, Fort Collins, CO, February 1989.

[6] J. T. Feo D. C. Cann and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[7] J. B. Dennis. Mapping programs for data parallel execution on the connection machine. Technical report, NASA RIACS, 1989.

[8] T. Gross and A. Sussmann. Mapping a single-assignment language onto the WARP systolic array. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, October 1987.

[9] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[10] C. Lee, S. Skedzielewski, and J. Feo. On the implementation of applicative languages on shared-memory, MIMD multiprocessors. In *Proceedings of the Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 188–197, New Haven, CT, September 1988. Available in *SIGPLAN Notices* 23, 9.

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

[12] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[13] R. R. Oldehoeft and D. C. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, pages 62–70, January 1988.

[14] R. R. Oldehoeft and J. R. McGraw. Mixed applicative and imperative programs. *Parallel Computing*, 13(2):175–191, 1990.

[15] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages*. PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[16] T. R. Richert. Efficient task management for SISAL. Technical Report 89-111, Computer Science Department, Colorado State University, July 1989.

[17] S. K. Skedzielewski and J. Glauert. IF1—an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[18] S. K. Skedzielewski and R. J. Simpson. A simple method to remove reference counting in applicative programs. Technical Report UCRL-100156, University of California Lawrence Livermore National Laboratory, November 1988.

[19] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in IF1. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 17–34. Springer-Verlag, New York, NY, September 1985.

[20] M. L. Welcome, S. K. Skedzielewski, R. K. Yates, and J. E. Ranelletti. IF2: an applicative language intermediate form with explicit memory management. Manual M-195, University of California Lawrence Livermore National Laboratory, November 1986.

Chapter 2

Syntactic Notations and Lexical Elements

In this chapter we describe the syntactic notations used to describe SISAL, and identify the basic lexical elements used to build programs.

2.1 Syntax Notations

To describe SISAL syntax in this manual, we use an extended BNF notation with the following conventions:

[stuff]	optional stuff
[stuff] · · ·	zero or more occurrences of stuff
{ stuff }	stuff
stuff1 | stuff2	stuff1 or stuff2

In contrast to these meta-symbols, the reserved words and special symbols that are part of the SISAL language appear in bold face type. The following productions illustrate some of the common combinations.

decldef-part	::=	decldef [; decldef] · · · [;]
module	::=	{ **program** | **module** } module-id
		[import-decl | type-def | function-def] · · ·
		end { **program** | **module** }

The first production describes a non-empty list of decldef's separated by semicolons. Note that a semicolon can optionally end the list, and the left bracket, right bracket, and centered ellipsis are part of the BNF meta-language and not SISAL. The second production describes the syntax of a module, which begins with either **module** or **program**, contains zero or more import-decl's, type-def's, or function-def's in any order, and ends with either **end module** or **end program**.

2.2 Lexical Elements

SISAL programs are written using the ASCII character set. However, no "control" characters other than tab and newline are allowed within the source. The basic program elements are:

- Operation and punctuation symbols
- Integer, real, complex, character and string constants
- Reserved words
- Names
- White space
- Comments

We describe each in turn.

2.2.1 Operation and Punctuation Symbols

The operation and punctuation symbols are:

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

2.2.2 Constants

The syntax of constants in SISAL is given by the following:

constant	::=	**false**
	|	**true**
	|	**nil**
	|	integer-constant
	|	real-constant
	|	double-constant
	|	complex-constant
	|	double_complex-constant
	|	character-constant
	|	character-string-constant
	|	**error** [type-spec]

An integer constant is a non-empty sequence of decimal digits without a decimal point. The type of an integer constant is integer.

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

- a decimal point,
- an exponent field,
- a decimal point followed by an exponent field,
- a decimal point followed by an integer constant,
- a decimal point followed by an integer constant followed by an exponent field.

An exponent field begins with one of the letters 'E', 'e', 'D', or 'd' optionally followed by a sign, and ends with an integer constant. A real constant without an exponent field, or with an exponent field beginning with either 'E' or 'e' is of type real. A real constant with letter 'D' or 'd' in its exponent field is of type double.

A complex constant is a comma-separated pair of numeric constants enclosed in parentheses. If neither component is of type double, the result has type complex. If at least one has type double, the result is of type double_complex.

A character constant is a single character enclosed in single quotes. A character string 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 of character, where the low bound is one and the high bound is the number of characters in the string.

Using notation taken from the C programming language, a character constant or string may contain any control character. 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	For use in string constants
\'	Apostrophe	For use in a character constant

A backslash followed by any other character besides the first of three consecutive octal digits, is simply the character itself; that is, \s yields the character s. A backslash followed by three octal digits represents the character whose numeric ASCII value is the octal value. For example, \101 is the character A. This format also allows the insertion of control characters into character constants and strings. For example, \011 is a horizontal tab. Although each of the above sequences looks like one or more characters, each represents exactly one.

2.2.3 Reserved Words

A reserved word always has special meaning, and its use is context dependent. The reserved words in SISAL are:

array	**at**	**boolean**	**case**
character	**complex**	**cross**	**default**
dot	**do**	**double**	**double_complex**
else	**elseif**	**end**	**error**
false	**foreign**	**for**	**forward**
from	**function**	**if**	**imag**
inout	**integer**	**interface**	**in**
is	**let**	**module**	**nil**
null	**of**	**otherwise**	**out**
program	**real**	**record**	**returns**
state	**stream**	**suffix**	**then**
to	**true**	**type**	**when**
while	**union**	**unless**	**until**

2.2.4 Names

A name is a sequence of letters, digits, and underscores, of which the first must be a letter. It can be an identifier for a module, language, function, reduction, type, union tag, record field, or value. Each has its own mechanisms of interpretation, so one can often use a name without conflict for several purposes. For example:

type NewComplex = **record** [Re, Im : **real**] ;
type Im = **boolean** ;

is a legal set of type definitions, even though the name Im appears twice—context information helps distinguish them. A name cannot be a reserved word. It can be of any length.

Upper and lower case letters in names, reserved words, and predefined functions are not distinguishable.

2.2.5 White Space

Space, tab, and newline characters are "white space" in SISAL source. They are equivalent, except when delimiting comments. One must use white space between adjacent constants, names, or reserved words to prevent misinterpretation. For example, "ifpthen3else4endif" is not equivalent to "**if** p **then** 3 **else** 4 **end if**." White space may not appear within a number or between the characters of a two character operation symbol such as `>=`.

2.2.6 Comments

A comment in SISAL source begins with a percent sign and ends with a newline character. If the character immediately following a percent sign is a dollar sign, the comment is a compiler directive, or pragma. See Appendix B for a description of pragma syntax and semantics. Comments are equivalent to white space. Examples of names and constants follow:

```
ABC3_Q ABC3_q                    % The same name twice
34                               % Integer constant
0.3141593E1                      % Real constant
0.3141593D1                      % Double constant
2.7182                           % Real constant
(1.5, -6.4E-5)                   % Complex constant
(1.5D0, -6.4e-5)                 % Double complex constant
'%'                              % Character constant
"abc\"\ndef" "abc\"\nd\ef"       % The same string twice
```

Chapter 3

Scalar Types and Values, and LET Scopes

This chapter introduces type definitions and specifications; describes SISAL's scalar types, values, and operations; presents the syntax of SISAL expressions and defines the concept of arity; introduces value names; and describes the **let** construct.

The inputs and outputs of SISAL expressions and functions are values. The value domain of SISAL is the entire collection of values that a SISAL program can receive and produce. This domain is subdivided into disjoint subdomains that are the data types of SISAL. There are basic types that include the familiar scalar values of computation, and compound types that include arrays, streams, records, and unions. In addition, SISAL supports function types.

3.1 Type Specification and Conformity

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

Syntax:

type-spec	::=	basic-type
	|	compound-type
	|	type-set
	|	[interface-id .] type-id
	|	**type** (expression)
basic-type	::=	**boolean**
	|	**character**
	|	**integer**
	|	**real**
	|	**double**
	|	**complex**

	|	**double_complex**
	|	**null**
	|	**foreign** language-id **function**
compound-type	::=	array-spec
	|	stream-spec
	|	record-spec
	|	union-spec
	|	function-spec
type-set	::=	type-spec [{ + | - } type-spec] $\cdots$

For a basic type, the specification is simply the type's name (a reserved word). For a compound type, the specification consists of the name of the compound type (a reserved word) followed by additional information. Two type specifications conform (are identical) if they derive compound types that conform, or derive the same scalar type. The compound types and their conformity rules, values, and associated operations are discussed in later chapters. A typeset is a set of alternative type specifications, and is described in Chapter 9.

3.2 Type Definitions

Type definitions introduce type names used as type specifications.

Syntax:

type-def	::=	**type** type-id [= type-spec] [;]
	|	**type state** [;]
type-id	::=	name

A type definition without a defining specification, and the special type **state**, appear only in a program structure called an interface, and are discussed in Chapter 10.

Within its scope, a type name (type-id) may be used anywhere a type specification is allowed. However, a type name cannot be referenced before its definition, unless it is referenced within the union specification (refer to Chapter 7). The scope of a type name is its defining construct minus any embedded construct that redefines it. That is, each type name can represent only one type specification throughout its scope of reference. Note that a type name can be used as any other kind of name without conflict.

3.3 Scalar Types and Operations

Each scalar data type has its own domain of values, and is characterized by the set of operations that create and transform values in that domain. SISAL also provides operations to convert values of one domain into those of another.

The operands for the operations described below may be scalars, arrays of scalars, or streams of scalars. For example, addition applied to two arrays of scalars will produce a new array containing the sum of their respective elements. Refer to Chapters 4 and 5 for further discussion of array and stream operations. In this chapter, we only consider the scalar data types.

3.3.1 Boolean Values and Operations

The proper elements of the boolean data type are **true** and **false**. The boolean operations are:

Operation	Notation	Functionality
And	P & Q	boolean,boolean → boolean
Or	P | Q	boolean,boolean → boolean
Not	~P	boolean → boolean
Equal	P = Q	boolean,boolean → boolean
Not equal	P ~= Q	boolean,boolean → boolean

3.3.2 Character Values and Operations

The ASCII characters are the proper elements of the character data type. Relational operators on characters are defined in terms of the comparison between their numeric representations in the ASCII character set. The operations are:

Operation	Notation	Functionality
Equal	C = D	character,character → boolean
Not equal	C ~= D	character,character → boolean
Greater than	C > D	character,character → boolean
Less than	C < D	character,character → boolean
Greater or equal	C >= D	character,character → boolean
Less or equal	C <= D	character,character → boolean

3.3.3 Numeric Values and Operations

The numeric types supported by SISAL include integer, real, double, complex, and double_complex. The proper elements of the integer data type are the whole numbers between implementation dependent limits. The proper elements of the real data type are the floating point numbers between implementation dependent limits. The proper elements of the double data type are also the floating point numbers between implementation dependent limits; however, the number of mantissa digits must be greater than or equal to the number of mantissa digits of the values in the real domain. The proper elements of the complex data type are pairs of real values. The proper elements of the double_complex data type are pairs of double values. The following numeric operations are supported in SISAL:

Operation	Notation	Functionality
Addition	X + Y	num,num → num
Subtraction	X - Y	num,num → num
Multiplication	X * Y	num,num → num
Division	X / Y	num,num → num
Exponentiation	X ** Y	num,num → num
Negation	- X	num → num
Equal	X = Y	num,num → boolean
Not equal	X ~= Y	num,num → boolean
Greater	X > Y	num,num → boolean
Less	X < Y	num,num → boolean
Greater or equal	X >= Y	num,num → boolean
Less or equal	X <= Y	num,num → boolean

The term "num," used to describe the operand and result data types in the functionality column above, represents types integer, real, double, complex, and double_complex. However, of the relational operations, only = and ~= are defined for complex and double_complex values. If the operands of a numeric operation have the same type, then the resulting value has that type. If the operand types are different, the operation will be applied after its operands have been implicitly converted, as necessary, to the higher type that exists between the operands (this is the type of the result).

The following table summarizes the result types for all the above operations (except unary minus, which yields the same type as its operand).

Left Operand	Right Operand: integer	Right Operand: real	Right Operand: double	Right Operand: complex	Right Operand: double_complex
integer	integer	real	double	complex	double_complex
real	real	real	double	complex	double_complex
double	double	double	double	double_complex	double_complex
complex	complex	complex	double_complex	complex	double_complex
double_complex	double_complex	double_complex	double_complex	double_complex	double_complex

Among relational operations, only `=` and `~=` apply to complex and double_complex operands. Note that an integer exponent with a base of another type is not necessarily converted to the other type before the operation. If the base is zero and the exponent is nonzero, the result is zero (of the appropriate type). If the base is nonzero and the exponent is zero, the result is one. If both are zero, the result is an error (See Chapter 11).

Here is an example of implicit type conversion:

```
4 + 2.0 + A
```

where `A` is of type double. First `4` is converted to real (4.0). Then `4.0` is added to `2.0` using real arithmetic. Then 6.0 is converted to double and added to A to form a double result. Hence the expression is of type double.

The following comments apply to values of the complex and double_complex types.

- Complex values are written as a parenthesis-enclosed, comma-separated pair of expressions:

  complex-gen ::= (expression , expression)

  The first expression is the real part, and the second is the imaginary part. This was introduced in Chapter 2 when discussing complex constants.

- The parts of a complex value are accessible via dot notation:

  complex-ref ::= primary . { **imag** | **real** }

3.3.4 Predefined Functions

These are the predefined functions of SISAL, along with a brief description of their operation:

abs	and	bit_size	conj	empty
eor	floor	ior	limh	liml
max	min	mod	not	rotate
shift	size	transpose	trunc	

abs: Accepts a numeric parameter and returns its magnitude. Parameter and result types coincide, with the exception that complex parameter types yield corresponding real type results.

conj: Accepts a complex or double_complex value and produces the complex conjugate value of the same type.

empty: Accepts a stream parameter; see Chapter 5.

floor: Produces the largest integer value not greater than its real or double parameter.

limh, liml, size, transpose: Accept array parameters; see Chapter 4.

max, min: Accept two or more integer, real, or double parameters, and return the greatest or least value respectively. The type of the result is the highest numeric type among the parameters.

mod: Accepts two integer parameters and produces the modulus. If Z is the result of mod(X,Y), then

- $0 \leq \text{abs}(Z) < \text{abs}(Y)$,
- the sign of Z equals the sign of Y, and
- there exists an integer M with $X = M \times Y + Z$.

trunc: Accepts a real or double parameter and returns the integer that results from removal of the fractional part of the parameter.

and, ior, eor: Accept two or more integer parameters, and return bitwise operations and, inclusive or, and exclusive or respectively.

not: Accepts an integer parameter, and returns the bitwise complement value.

shift: Accepts two integer parameters, a value to be treated bitwise, and a shift count. The number of bits to shift is given by the maginitude of the shift count, and its sign determines the direction: positive for left shift, and negative for right shift. In either case, vacated bits are filled with zero bits.

rotate: Accepts an integer value to be treated bitwise, and a bit count. In this case a positive count rotates the bits left in a circular fashion, and a negative value rotates the bits circularly right.

bit_size: Accepts no parameter; returns the number of bits in the representation of the integer data type.

Others are more specialized, and are described in appropriate later sections. Some of these functions also accept arrays or streams of the scalar parameters described here. See Chapters 4 and 5 for details.

3.3.5 Type Values and Operations

SISAL has limited facilities for inquiring about the types of expressions, and for comparing one type specification with another. The nature of the language guarantees that all type inquiries may be known concretely before execution begins, and so also are any results based on these operations. These facilities are primarily useful with values whose types are defined with typesets (see Chapter 9).

The operations on type values are:

Operation	Notation	Functionality
Inquiry	**type**(X)	expression → type-spec
Equality	X = Y	type-spec,type-spec → boolean
Inequality	X ~= Y	type-spec,type-spec → boolean

The form "**type**(expression)" has as value the type of the expression. As an alternative in the definition of type specification, this form may appear wherever the other alternatives for type specification may appear, as long as the type of the expression may be inferred in that context.

Type specifications may be compared using relational operators = and ~=. Selection expressions may be controlled by type comparison or inquiry (see Chapter 8).

3.4 Type Conversion

In SISAL, the translator can determine the type of value produced by an operation from the properties of its operands; hence the type of an expression is inferable. Types must match the types required by their context of use. For example, a parameter to a function must match the parameter type shown in the function's definition.

Besides the implicit type conversion to higher types for the numeric operations, and the predefined functions floor and trunc already described, SISAL provides several explicit type conversion operations:

Operation	Notation	Functionality
real-to-integer	**integer**(X)	real → integer, double → integer
integer-to-real	**real**(J)	integer → real
	double(J)	integer → double
real-to-real	**real**(X)	double → real
	double(X)	real → double
character-to-integer	**integer**(C)	character → integer
integer-to-character	**character**(J)	integer → character
complex-to-complex	**complex**(Y)	double_complex → complex
	double_complex(Y)	complex → double_complex
real-to-complex	**complex**(X)	real → complex
	double_complex(X)	double → double_complex
integer-to-complex	**complex**(J)	integer → complex
	double_complex(J)	integer → double_complex

An integer, real, or double parameter to a **complex** or **double_complex** conversion operation is the real part of the resulting value; the imaginary part is zero.

The **integer**(X) operation yields **floor**(0.5+X).

The **integer**(C) operation yields the ASCII code for character C.

The **character**(J) operation is the inverse of **integer**(C).

3.5 Operators and Precedence

SISAL expression operators obey the customary precedence rules. From highest to lowest they are:

```
Unary arithmetic    +   -
Exponentiation      **
Multiplication      *   /
Binary addition     +   -
Concatenation       ||
Relation            <= >= ~= = < >
Boolean negation    ~
Conjunction         &
Disjunction         |
```

Of these operators, only ** is right associative. A programmer can always parenthesize an expression to override SISAL's default precedence and associativity rules.

3.6 Expressions

The expression is the basic syntactic unit of SISAL and denotes a list of values of some types. The arity of an expression is the size of its list of values. Two expressions conform if they have the same arity and have values of the same types. SISAL is designed so that program inspection can yield the arity and types of an expression, and hence the conformity of two expressions or two expression lists. The simplest type of expression of arity one is a constant, a value name (discussed later in this chapter), 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. SISAL's expression syntax follows:

```
expression   ::=  expression binary-op expression
              |   primary
unary-op     ::=  + | - | ~
binary-op    ::=  < | <= | > | >= | = | ~= | + | - | | | * | ** | / | & | ||
primary      ::=  ( expression )
              |   unary-op expression
              |   [ interface-id . ] function-id
              |   reduction-id
              |   value-id
              |   constant
              |   invocation
              |   error-test
              |   union-ref
              |   union-test
              |   union-gen
              |   record-ref
              |   record-gen
```

	|	stream-ref
	|	stream-gen
	|	array-ref
	|	array-gen
	|	array-update
	|	complex-ref
	|	complex-gen
	|	conditional-exp
	|	let-exp
	|	case-exp
	|	for-exp
	|	type-spec
invocation	::=	primary ([[expression] [, [expression]] · · ·])
error-test	::=	**is error** (expression)
union-test	::=	**is** tag-id (expression)
union-ref	::=	primary . tag-id
union-gen	::=	**union** type-spec [tag-id : expression [;]]
	|	primary [tag-def [;]]
tag-def	::=	tag-id [. tag-id] · · · : expression
record-ref	::=	primary . field-id
record-gen	::=	**record** type-spec [field-def [; field-def] · · · [;]]
	|	primary [field-def [; field-def] · · · [;]]
field-def	::=	field-id [. field-id] · · · : expression
size-descriptor	::=	[value-id **in**] triplet [, [value-id **in**] triplet] · · ·
triplet	::=	[expression] . . [expression] [. . expression]
selector	::=	selector-part [, selector-part] · · ·
		[{ diag-spec [, diag-spec] · · · }]
selector-part	::=	expression | [value-id **in**] triplet
diag-spec	::=	value-id **dot** value-id [**dot** value-id] · · ·
array-ref	::=	primary [selector]
array-update	::=	primary [update-part [; update-part] · · · [;]]
update-part	::=	selector : expression-list
array-gen	::=	**array** type-spec
		[[[size-descriptor :] array-part [; array-part] · · · [;]]]
array-part	::=	[placement] expression-list
placement	::=	[{ selector | **otherwise** }]
stream-ref	::=	primary [{ triplet | expression }]
stream-gen	::=	**stream** type-spec [[stream-part [, stream-part] · · ·]]
stream-part	::=	expression
	|	triplet
expression-list	::=	expression [, expression] · · ·

The array and stream operations are discussed in Chapters 4 and 5 respectively; the record and union operations are discussed in Chapter 7; and the **for**, and **if** and **case** constructs are discussed in Chapters 6 and 8 respectively. Chapter 9 discusses function definitions and invocations. The **let** expression is discussed in the next section.

3.7 Value Names and LET Construct

3.7.1 Value Names

A value name is a name that denotes a single computed value of a specific type. Every value name is introduced either in the header of a function definition (the value name is a formal parameter of the function being defined), or in a construct such as a **let** or **for**. Regardless of the source, each value name has a type and scope, and a unique value throughout its scope. Its value, however, depends on the values present during its definition. Like type names, the scope of a value name is its defining construct minus any embedded constructs that redefine it. Again, type and value names cannot conflict.

The type of a value name *may* be part of its definition, and each value name must be defined before use. The definitions may be mixed in any order as long as these requirements are met.

Examples:

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

Several value names may be defined in the same assignment. The number and types of the names must conform to the expressions on the right hand side.

Examples:

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

Several value names may be declared and defined at once. 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.

3.7.2 The LET Construct

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

Syntax:

let-exp	::=	**let** decldef-part **in** expression-list **end let**
decldef-part	::=	decldef [; decldef] · · · [;]
decldef	::=	decl [, decl] · · · := expression-list
decl	::=	decl-id [, decl-id] · · · [: type-spec]
decl-id	::=	value-id
	|	function-header
value-id	::=	name

Each value name can be defined in a **let** just once. The expression list following the reserved word **in** is in the scope of all the defined value names, and can use them. The values of these expressions are the results of the **let** construct. Chapter 9 discusses function definition (the second rule of the decl-id production).

The scope of each value name introduced in the **let** construct is the entire construct less any inner constructs that reintroduce the same value name. However, a value name must not be used in definitions preceding its own definition. All scopes for value names not introduced in a **let** construct pass into that construct. Hence, if the scope of a value name (introduced by an outer construct) includes a **let** construct and that value name is not reintroduced, it may be referred to freely within the construct.

Example:

```
let
  X := 3.0; % this semicolon is required!
  A := X*G; % this semicolon is optional!
in
  A*X, X
end let
```

This expression has arity two, yielding `3.0*G*3.0` and `3.0` as its values. Value name `X` must be defined before `A`. The scope of both `X` and `A` is the entire construct. If the construct had appeared within another scope that defined `X` or `A`, then the outer definition of the reused name would disappear within this **let** construct. Note that the semicolon before the reserved word **in** is optional. In SISAL, semicolons act as separators and are not required before reserved words.

Since a value name may not be used until after it has been defined, and can be defined only once in a scope, it may not appear in its own definition. An occurrence in the defining expression of the name being defined is legal if the name is being inherited from a surrounding scope. For example, the following yields `4`.

```
let A := 3 in
             let A := A + 1 in A end let
end let
```

Chapter 4

Arrays

This chapter describes SISAL's array data type and associated operations. An array is a one-dimensional or multidimensional collection of homogeneous values indexed by simple integer expressions (for single elements) or *triplets* (for subarrays).

triplet ::= [expression] . . [expression] [. . expression]

A triplet is a syntactic entity used in several contexts for arrays and streams; it also plays a role in the `for` expression (see Chapter 6). In different usages, some components of a triplet may be required or disallowed. In one version, up to three integer-valued expressions may be present. The first gives a lower bound, the second an upper bound, and the third a *stride* that defines the distance between consecutive values. If the first or second expression is missing, the lower or upper bound from context is sometimes used. If the third expression is missing, 1 is always used. Note that in all contexts, the sign of a stride defines a triplet's "direction," and a stride can never be zero.

In an alternate version, a single expression may evaluate to a one-dimensional array of integers. Each integer is used in turn to select an array element, and hence each must be a legal value in the dimension in which it appears. We will refer to these as *vector subscripts* and mention the three contexts in which they may be used in appropriate sections below.

In SISAL a name being defined as an array value may not appear in the expression defining the name. That is, arrays are not defined recursively. Chapter 12 discusses some issues on this subject.

4.1 Type Specification, Conformity, and Shape

The array type specification gives the array's dimensionality and component type, which can be any valid SISAL type. The syntax follows:

array-spec ::= **array** [dim-spec] **of** type-spec

dim-spec	::=	`[ ..` [`, ..`] ··· `]`

The dimension specification is a bracket enclosed list of dot-pairs. It is optional, and its absence specifies a one-dimensional array. If the dimension specification is present, it declares a one-dimensional or multidimensional array, depending on the number of dot-pairs in the list. Examples follow:

```
type Arr1 = array of double;          % 1-Dimensional
type Arr2 = array [..] of double;     % 1-Dimensional
type Arr3 = array [..,..] of double;  % 2-Dimensional
```

Two array types *conform* if they have the same dimensionality and their component types conform. For example, `Arr1` and `Arr2` above conform, but `Arr2` and `Arr3` do not. Note also that a one-dimensional array whose elements are one-dimensional arrays is never conformable with a two-dimensional array. Extents in each dimension are not given for array types, and are not part of type conformity.

4.2 Generation

SISAL has two ways for building arrays. The first is via the **for** expression, discussed in Chapter 6. The second is using the array generator:

array-gen	::=	**array** type-spec
		`[` [[size-descriptor :] array-part [; array-part] ··· [;]] `]`
size-descriptor	::=	[value-id **in**] triplet [, [value-id **in**] triplet] ···

where array-part is defined by

array-part	::=	[placement] expression-list
placement	::=	`[` { selector | **otherwise** } `]`
selector	::=	selector-part [, selector-part] ···
		[{ diag-spec [, diag-spec] ··· }]
selector-part	::=	expression | [value-id **in**] triplet
diag-spec	::=	value-id **dot** value-id [**dot** value-id] ···

The type specification, which identifies the component type of the array being generated, is required. The size descriptor is required for multidimensional arrays. A semicolon separated sequence of array-parts defines the array's components and specifies their placements within the array.

A size descriptor is a comma-separated list of triplets. Each triplet gives the lower and upper bound of the associated dimension. In this context the lower bound is optional, defaulting to 1; the upper bound is not optional and must be at least as large as the lower bound; and the stride is always assumed to be 1 (that is, its specification is not allowed).

Each array-part can be optionally preceded by a placement specifier defining where the associated components should be placed within the array. If a placement specifier is provided, one must be provided for each part, and the generator must provide a size descriptor as well. (Note that the resulting array need not be multidimensional.) In the absence of a size descriptor, however, there can be at most one array part, and the array must be one-dimensional. If the generator defines a multidimensional array, it must include a size descriptor and each array part, if there is more than one, must have a position specifier. The placement specifiers must together cover exactly the space defined by the size descriptor, and their intersections must be empty. If this is not the case, an error results (See Chapter 11). As this implies, each selector provided as a position specifier must have a selector-part for each dimension of the array (the **otherwise** option is discussed shortly). The expressions within the selector-part of a placement specifier must yield integers, or be a vector subscript specifying a set of legal subscript values in this dimension.

Consider the following examples:

```
let
    A1 := array integer [1..4: 10, 20, 30, 40];
    A2 := array real [2..3,1..2: [2,..] array real [1.0, 2.0];
                                 [3,..] array real [3.0, 4.0] ];
    A3 := array integer [1..4: [2] 20; [1] 10; [3] 30; [4] 40 ];
    A4 := array real [2..3,..2: [2,1] 1.0; [2,2] 2.0;
                                [3,1] 3.0; [3,2] 4.0 ];
    A5 := array integer [ 10, 20, 30, 40 ];
    A6 := array real [];
in
    A1, A2, A3, A4, A5, A6
end let
```

Here arrays `A1`, `A3` and `A5` are identical, as are arrays `A2` and `A4`. Array `A6` is an empty one-dimensional array of reals, where the lower bound is 1 and upper bound is zero.

Using triplet notation in a placement specifier (selector-part), one can map the values of an array part into a geometrically-defined subset of the array being generated. Here the placement specifier defines an arithmetic sequence of subscript positions formed from the cross product of the triplets used. In each triplet, the first integer expression is the starting subscript value in the dimension, and defaults to the generated array's lower bound in the dimension. The second integer expression gives the last subscript value in the dimension, defaulting to the generated array's upper bound in the dimension. The third integer expression is the *stride*, defaulting to 1; it defines the direction of the triplet. For example, the following expression builds a copy of one-dimensional array `A` assuming it has exactly `N` real components:

```
array real [1..N: [1..] A]
```

(Observe that this is not a recommended way to obtain a copy of an array value; a simple assignment `B := A` suffices.) Because `A` is an array of the generated array's component type, it fills the area defined by the placement. In this context, the shape of the placement specifier must equal the shape of the part, as above. They have the same shape if they have the same number of dimensions and the same number of components in their corresponding dimensions. If the expression directly defines a component value of the resulting array, then as expected the value is replicated throughout the specified positions. For example, the following maps array `B` into the first row of a generated matrix `C`, and fills the second row with zeros:

```
B := array integer [1..2: 3, 4 ];
C := array integer [1..2,1..2: [1,..] B; [2,..] 0 ];
```

Note that the component type of `B` matches the component type of `C` and the placement specifier and `B` have the same shape, and `0` has the component type of `C`—integer. Assuming `A` is an array of reals, the following generator builds a one-dimensional array, each element of which is a copy of A:

```
array array of real [1..N: [1..] A ]
```

As a convenience, the values in a multidimensional array-part may be component values of the overall array, and listed in row-major order. They must be exactly sufficient to fill the entire array-part. For example, the following defines the same array as `C` above:

```
array integer [1..2,1..2: 3, 4, 0, 0 ]
```

A special placement specifier **otherwise**, which always accompanies an expression of the generated array's component type, allows the remainder of the generated array's values to be defined at once. The **otherwise** may appear only once in an array generator. For example, consider

```
D := array real [1..N, 1..N: [1,1..N-1] 12.3; [otherwise] 0.0 ]
```

Here `otherwise` is the union of `[2..,..]` and `[1,N]`.

The following example uses the third expression of a triplet to map constant `1` to each odd element of a vector and constant `0` to each even element:

```
array integer [1..N: [1..N..2] 1; [2..N..2] 0 ]
```

The triplets found within a placement specifier can be named and referenced in a **dot** diag-spec to describe diagonal components. For example, the following expression builds an identity matrix of type real.

```
array real [1..N,1..N:
            [ i in 1..N, j in 1..N {i dot j} ] 1.0;
            [ otherwise ] 0.0 ]
```

In this context, the scope of a triplet name is the defining array part minus all expressions of the part's placement specifier. Further, it is only visible to an expression defining an array item if the type of the expression matches that of the generated array's component, and the expression is the only expression in the part's expression-list.

In addition to allowing the specification of diagonal components, named triplets also provide a valuable shorthand notation for a variety of **for** expressions (see Chapter 6). For example, the following computations produce two identical Hilbert matrices.

```
X := array real [ i in 1..N, j in 1..N: [i,j] 1.0/(i+j-1) ];
Y := for i in [1..N] cross j in [1..N] do
        returns array[1..N, 1..N] of 1.0/(i+j-1)
     end for;
```

As shown, a named triplet can be introduced in a size descriptor. The scope of this name is the entire array generator except the size descriptor itself and any parts redefining the name. All triplet names introduced in a size descriptor must be unique, as must all triplet names introduced in a placement specifier.

In a placement part, one or more dimensions may be described by an expression that is in fact a vector subscript. This provides a facility for placing values into the overall array being generated in a non-geometric way. Each integer in the vector subscript is used in turn to describe an array position for a value. In generation, the subscripts derived from use of vector subscripts must specify unique array element positions. For example,

```
let
  U := array integer[ 1, 4, 2 ]
in
  array real[ 1..3, 1..4: [1, U] 11.1, 14.4, 12.2;
                          [2, U] 21.1, 24.4, 22.2;
                          [otherwise] 0.0 ]
end let
```

defines a rectangular array

$$\begin{pmatrix} 11.1 & 12.2 & 0.0 & 14.4 \\ 21.1 & 22.2 & 0.0 & 24.4 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

4.3 Subarray Selection

The syntax for subarray selection is found in this section.

array-ref ::= primary [selector]

The leading primary gives the array value being accessed. The selector and its subparts were explained in the previous section, where each subpart can take one of the forms: a simple integer expression, triplet, named triplet, or vector subscript. If each selector part is a simple integer expression, then the result is a component of the array value. If at least one selector part is a triplet, named triplet, or vector subscript, then the result is a subarray whose dimensionality matches the number of such items used. However, if a **dot** diag-spec appears, all the dotted triplets together contribute one dimension to the result. The lower bound in each dimension of a selected subarray is 1. Examples follow:

```
let
  A  := array real [1..4: 1.0, 2.0, 3.0, 4.0 ];
  B  := array integer [1..3,2..4 : [1,..] 1, 2, 3;
                                   [2,..] 4, 5, 6;
                                   [3,..] 7, 8, 9 ];
  U  := array integer[ 1, 4, 2, 1 ]
in
  A[3], B[2,3..4], A[ U ]
end let
```

The expressions of the **in** part yield, respectively,

```
3.0
array integer [1..2: 5, 6]
array real [ 1.0, 4.0, 2.0, 1.0 ]
```

In selection, the elements of a vector subscript need not be unique.

The named triplet notation allows the selection of arbitrary diagonals. In this context, triplet names have scope limited to their selector's diagonal specifier. For example, assuming the definition of `B` above,

```
B[i in 1..3, j in 4..2..-1 {i dot j}]
```

and

```
B[i in 3..2..-1, j in 2..4..2 {i dot j}]
```

yield results `array integer [1..3: 3, 5, 7]` and `array integer [1..2: 7, 6]`. The above shows that the stride expression in a triplet allows the selection of non-adjacent values, and controls direction.

4.4 Update

SISAL's facilities for array updating are a subset of facilities for generation and subarray selection. The result of an update is a new array differing in the specified index positions or subarrays. The syntax is:

array-update	::=	primary `[` update-part $\big[$; update-part $\big]\cdots\big[$;$\big]$ `]`
update-part	::=	selector : expression-list

As in subarray selection, the leading primary is an array value. Either the shapes of the selector and the associated expression match, or the expression has the array's component type and is replicated as needed. As in generation, we require an empty intersection of the selectors, or the result is an error (See Chapter 11). Vector subscripts are allowed here, but may not result in a repeated array position for update.

The following example updates the third, and second through fifth elements of `A`, and also yields `A` itself.

```
let
    A := array integer [ 1..5: 1, 2, 3, 4, 5 ];
in
    A[3:0], A[2..5: 60, 70, 20, 10], A
end let
```

Note that the updates yield new arrays and do not affect the value of `A` (the third result of the **let** expression). The following example builds an array from `B` with the odd elements set to zero:

```
B[1..N..2: 0]
```

As in array generation and selection, named triplets can be used to replace diagonals of multidimensional arrays. In this context, their scope is their associated array update part minus the expressions comprising the selector. For example,

```
let
  A := array integer [1..2,-3..-2: [1,..] 1, 2; [2, ..] 3, 4 ]
in
  A, A[i in 1..2, j in -3..-2 {i dot j}: 0]
end let
```

yields `A` and the new matrix

```
array  integer [1..2,-3..-2: [1,..] 0, 2; [2,..] 3, 0 ]
```

4.5 Bounds Inquiry

4.5.1 Liml and Limh Functions

Predefined functions liml and limh return the lower and upper bounds, respectively, of a single dimension of an array. By default, they interrogate the first dimension. An optional second parameter (an integer) can specify the desired dimension of inquiry.

Using `A` above, the references

```
liml( A ), limh( A, 2 ), liml( A, 1 )
```

have values `1`, `-2`, and `1`.

4.5.2 Size Function

By default, the predefined function size returns the total number of elements composing its parameter, which may be of any dimensionality. An optional second parameter (an integer) can restrict the dimension of inquiry. The size of an empty array is zero. Again using `A` above,

```
size( A ), size( A, 1 ), size( A, 2 )
```

have values `4`, `2`, and `2`.

4.6 Concatenation

Infix operator `||` concatenates two one-dimensional arrays of conforming component types; it also works with streams (refer to Chapter 5). The result is a new array comprising the components of the first followed by those of the second. The lower bound of the result is 1.

For example, given

```
A1 := array integer [ 10 ];
A2 := array integer [ 20, 30 ];
A3 := array integer [ 40 ];
A4 := array integer [];
```

`A1||A4||A3||A2` yields a new array with lower bound 1 and 10, 40, 20, and 30 as its components.

4.7 Arrays and Scalar Operations

The scalar operations discussed in Chapter 3 work *element-wise* with value-conforming array operands whose component types are legal when the operation is applied to corresponding elements. Value conformance means type conformance with the same extents in corresponding dimensions. The lower bound of each dimension of the new array is 1.

4.7.1 Infix Operators

The infix scalar arithmetic operators `+`, `-`, `*`, `/`, `**`, and unary `+` and `-` also accept array operands, as do the relational operators `=`, `~=`, `>`, `>=`, `<`, and `<=` and the boolean operators `&`, `|`, and unary `~`. Both array operand values must be conformable. For convenience, one operand of a dyadic operator may be a scalar and used with each array element in the other operand. Regardless, the rule of implicit type conversion holds for the arithmetic operations.

The following expression exemplifies some of the operations listed above when applied to arrays:

```
let
    A1 := array integer [ 1, 2, 3 ];
    A2 := array integer [5..6,-1..0: [5,..] 4, 5;
                                     [6,..] 6, 7 ];
    A3 := array boolean [ true, false ];
in
    ~A3, A1 + A1, A2 * 2.0
end let
```

It yields three new arrays

```
array boolean [1..2: false, true ],
array integer [1..3: 2, 4, 6 ],
array real [1..2, 1..2: [1,..]  8.0, 10.0;
                        [2,..] 12.0, 14.0 ]
```

Notice that implicit type conversion was required to produce the third array.

4.7.2 Predefined Scalar Functions and Operations

Predefined functions abs, and, conj, eor, floor, ior, max, min, mod, not, rotate, shift, and trunc, and operations integer, real, double, complex, double_complex and character all work with array parameters as well as scalars. Like the infix operators discussed in the previous section, the and, eor, ior, max, min, mod, rotate and shift functions can take a mixture of scalar and array parameters. Also, the rule of implicit type conversion holds for min and max, as it does for the arithmetic operators.

The following expression illustrates the use of some of these functions:

```
let
    S1 := array integer [2..4: 12, 4, 35 ];
    S2 := array integer [ 8, 9, 3 ];
in
    double( S1 ), max( S1, S2, 15.0 ), mod( S1, 3 )
end let
```

It yields three new arrays

```
array double_real [1..3: 12.0D0, 4.0D0, 35.0D0 ],
array real [1..3: 15.0, 15.0, 35.0 ],
array integer [1..3: 0, 1, 2 ]
```

The predefined function transpose accepts an array of any dimensionality and component type and produces a transposed array with the order of dimensions reversed. For example,

```
let
    A := transpose( array T [1..10, -1..1, 5..15: ... ] )
in
    size(A,1), size(A,2), size(A,3)
end let
```

yields `11, 3, 10`.

The predefined array reduction operations are discussed in Chapter 9.

Chapter 5

Streams

This chapter describes SISAL's stream data type and associated operations. Like arrays, streams are collections of homogeneous values. Differing from array components, however, stream components are only accessible in sequence. However, the syntax of substream selection resembles random access. In SISAL a name being defined as a stream value may not appear in the expression defining the name. That is, streams are not defined recursively.

5.1 Stream Type Specification and Conformity

The stream type specification defines a stream's component type; like arrays, it can be any valid SISAL type. The syntax follows:

stream-spec ::= **stream of** type-spec

For example, `StrA` shown below is a stream of one dimensional arrays.

```
type ArrD = array [..] of double;
type StrA = stream of ArrD;
```

Two stream type specifications conform if their component types conform. For example, out of the following type definitions

```
type Type1  = real;
type Arr2   = array [..] of double;
type SType1 = stream of Arr2;
type SType2 = stream of array [..] of double;
type SType3 = stream of Type1;
```

only `SType1` and `SType2` conform.

5.2 Stream Generation

SISAL has two ways to build streams. The first is via the **for** expression, discussed in Chapter 6. The second is the stream generator:

stream-gen	::=	**stream** type-spec `[` [stream-part [, stream-part] ···] `]`
stream-part	::=	expression
	|	triplet
triplet	::=	[expression] . . [expression] [. . expression]

Each stream-part produces one or more components, where the order of presentation (read left to right) prescribes their positions within the resulting stream. The first part, or left-most, gives the first component; it resides at position one. The type specification following the **stream** reserved word describes the component type of the generated stream. All the stream parts must conform in type.

A stream-part can take one of two forms: an expression, or a triplet. Either of these forms can appear within the same generator. The expression form yields a single component. The triplet form, which produces an arithmetic progression of integer values, can yield zero or more components.

The first two expressions of the triplet form give the lower and upper bounds of an arithmetic progression. The third expression gives the stride of the progression and defines its direction. If the first expression is missing, it defaults to 1. If the second is missing, it defaults to positive or negative infinity depending on the sign of the stride. If the stride is missing, it defaults to 1.

Examples:

```
stream integer [1..]                  % 1, 2, 3, ...
stream integer [1..4]                 % 1, 2, 3, 4
stream integer [10..1..-3]            % 10, 7, 4, 1
stream integer [..]                   % same as first example
stream integer [1..-3..1]             % an empty stream
stream integer [0....-1]              % 0, -1, -2, ...
stream integer [N-1, 1..3, 0, 0, 0]   % N-1, 1, 2, 3, 0, 0, 0
```

5.3 Substream Selection

The syntax for substream selection is:

stream-ref	::=	primary `[` { triplet | expression } `]`

The triplet form selects a substream, and the expression form selects a single stream component. Regardless, the indexing expressions must yield integers. A one-dimensional array of positive integers may select elements as well.

Substream selection results in a new stream that comprises possibly fewer components than the original. The first two expressions of the triplet give the lower and upper selection bounds. If the first is missing, it defaults to 1. If the second is missing, it defaults to n, where n is the size of the original stream. That is, `S[..]` is an identity operation; it yields a new stream that is identical to `S`. The third expression of the triplet is the selection stride; its default value is 1.

Remember that the first component of any stream (including that resulting from substream selection) is at position 1. Unlike arrays, this can never change.

The following **let** construct exemplifies both substream selection and stream inquiry:

```
let
    S := stream integer [40, 50, 60, 70, 80, 90, 100 ];
    N := 7;
in
    S[2..], S[1], S[4..N-1..2], S[N-2]
end let
```

The first expression (`S[2..]`) yields a new stream comprising 50, 60, 70, 80, 90, and 100. The second expression yields 40; the third expression yields a new stream comprising 70 and 90; and the fourth expression yields 80, the fifth component of `S`.

We mentioned at the beginning of this chapter that stream inquiry and substream selection give the illusion of random access. However, the substream selection and inquiry syntax is shorthand for sequences of "first", "rest," and generate operations, where "first" is `S[1]` and "rest" is `S[2..]`. For example, an unabbreviated expression for `S[4]` is:

```
let
    TS1 := S[2..]
    TS2 := TS1[2..];
    TS3 := TS2[2..];
in
    TS3[1]
end let
```

An unabbreviated expression for `S[3..4]` is:

```
let
    TS1 := S[2..]
    TS2 := TS1[2..];
    V3  := TS2[1];
```

```
        TS3 := TS2[2..];
        V4  := TS3[1]
    in
        stream integer [ V3, V4 ];
    end let
```

5.4 Stream Empty Inquiry Function

Empty is a predefined function that only works with streams. It takes a single stream parameter of arbitrary component type, and returns **true** if the stream has no components, else it returns **false**. This is useful in writing iterative `for` expressions that consume streams. For example, `empty(S1)` yields **false** if `S1` is `stream integer[1,2,3]`.

5.5 Stream Concatenation

The concatenate operator (`||`) works with streams as well as one-dimensional arrays. The component type of each operand must conform. The result is a new stream comprising the components of the first followed by those of the second.

For example, assuming

```
    S1 := stream integer [ 10 ];
    S2 := stream integer [ 20, 30 ];
    S3 := stream integer [ 40 ];
    S4 := stream integer [];
```

then `S1||S4||S3||S2` yields a new stream comprising 10, 40, 20, and 30.

5.6 Streams and Scalar Operations

The scalar operations defined in Chapter 3 can also take stream operands. The component type of each stream must conform to the type expectations of the particular operation. Each of the operations, working in an element-wise fashion, yields a new stream of the appropriate type. Both streams must have the same number of values.

5.6.1 Infix Operators

The infix scalar operators can take stream operands. This includes the arithmetic operators `+`, `-`, `*`, `/`, `**`, and unary `+` and `-`; the relational operators `=`, `~=`, `>`, `>=`, `<`, and `<=`; and the boolean operators &, |, and unary ~. Both operands of the dyadic operators listed above need not be streams; scalar parameters are conformable to stream parameters having compatible components. The scalar is treated as a stream whose components are all equal to the scalar. Regardless of operand type, the rule of implicit type conversion holds for the arithmetic operations.

The following expression exemplifies some of the operations listed above when applied to streams:

```
let
    S1 := stream integer [ 1, 2, 3 ];
    S2 := stream integer [ 4, 5, 6 ];
    S3 := stream boolean [ true, false ];
in
    ~S3, S1 + S2, S2 * 2.0, S1**S1
end let
```

It yields four new streams equivalent to

```
stream boolean [ false, true ],
stream integer [ 5, 7, 9 ],
stream real [ 8.0, 10.0, 12.0 ],
stream integer [ 1, 4, 27 ]
```

respectively. Notice that implicit type conversion occurred to produce the third stream.

5.6.2 Predefined Scalar Functions and Operations

Predefined functions abs, and, conj, eor, floor, ior, max, min, mod, not, rotate, shift, and trunc, and operations integer, real, double, complex, double_complex and character all work with stream parameters as well as scalars. Like the infix operators discussed in the previous section, the and, eor, ior, max, min, mod, rotate and shift functions can take a mixture of scalar and stream parameters. Also, the rule of implicit type conversion holds for min and max, as it does for the arithmetic operators.

The following expression illustrates the use of some of these functions:

```
let
    S1 := stream integer [ 12, 4, 35 ];
    S2 := stream integer [ 8, 9, 3 ]
in
    double( S1 ), max( S1, S2, 15 ), mod( S1, 3 )
end let
```

It yields four new streams, each equivalent to

```
stream double [ 12.0D0, 4.0D0, 35.0D0 ],
stream integer [ 15, 15, 35 ],
stream integer [ 0, 1, 2 ]
```

respectively.

Chapter 9 describes the predefined stream reduction operations.

Chapter 6

FOR Expressions

This chapter describes SISAL's **for** expression, which uses either distribution of control or iteration to produce one or more values. In some sense, it resembles iteration in conventional languages, but retains single assignment and functional semantics, and is itself an expression that can appear within other expressions. The syntax follows:

for-exp	::=	[for-top]
		for-body
		for-bottom
for-top	::=	**for** in-exp-list [; [decldef-part]] [for-test]
	|	**for** decldef-part [for-test]
	|	for-test
for-body	::=	**do** [decldef-part] [for-test]
for-bottom	::=	**returns** return-exp [, return-exp] · · ·
		end { **for** | **while** | **until** | **do** }
for-test	::=	{ **while** | **until** } expression
return-exp	::=	**stream of** expression [filter]
	|	expression [filter]
	|	**suffix** (value-id)
	|	**array** [[[size-descriptor]]] **of** expression
		[filter | **at** [at-exp [, at-exp] · · ·]]
filter	::=	**when** expression
	|	**unless** expression
at-exp	::=	expression | ..
in-exp-list	::=	in-exp [**dot** in-exp] · · ·
	|	in-exp [**cross** in-exp] · · ·
in-exp	::=	value-id **in** expression [**at** [index-id-list]]
	|	value-id **in** [triplet]
index-id-list	::=	{ .. | value-id } [, { .. | value-id }] · · ·

Note that the scope of any value name defined in an enclosing construct is passed into the **for** expression, but the value name may be re-established within the construct. The scope of any value names introduced in the **for** expression is the entire construct textually below its definition. That is, the **for** expression like all other constructs in SISAL must obey the "definition before use" rule.

The **for** expression contains separate sections for value name initialization and termination testing, distributed or repetitive action (the body), and result calculation. The general strategy is to introduce some value names, and distribute them and all other names visible to the **for** expression into one or more instances of the body, or re-establish one or more of them during successive body executions. When the distributed body invocations complete, or all termination criteria are met, or both, the sequences of values produced by the body executions are passed to the results section for packaging. The values can be packaged into arrays or streams, or combined using predefined reduction operations (discussed in Chapter 9). It is possible to return the last value of a sequence, or the last value meeting a particular criterion.

The **for** expression has three parts: a top, body, and bottom. The top part establishes zero or more value names (each defined once) and optionally gives the body distribution or iteration control logic. Providing control in the top part can prevent execution of the body altogether, defining a zero trip iteration form. Note that the top part is optional, as it is possible to provide a termination condition that executes after each body execution, defining a post-test iteration form. It is possible to specify body distribution control and iteration control in the same **for** expression, resulting in an iterative computation.

The body part (beginning with **do**) defines the actions to be carried out in either a distributed or iterative manner. In the iterative form, the body can re-establish carried values for the next iteration (discussed in more detail below).

The bottom or returns part, which begins with reserved word **returns**, packages the values produced in the **for** body. If the body never executes, neither does the bottom part. In this case, the resulting values are the default values of the various returns parts (discussed in Section 6.2). Note that the closing reserved word of the **for** expression must match the reserved word introducing the **for** expression: (**for**, **while**, **until**, or **do**).

The following example illustrates a pre-test, iterative **for** expression, calculating π:

```
for
    Approx := 1.0; Sign := 1.0; Denom := 1.0; i := 1
while i <= Cycles do
    Sign   := -Sign; Denom := Denom + 2.0;
    Approx := Approx + Sign / Denom;
    i      := i + 1
returns Approx * 4.0
end for
```

It returns the last value of `Approx` after multiplying it by `4.0`.

The next example presents a **for** expression with body distribution control instead of iteration control as illustrated in the previous example. This expression also calculates π, producing the same result if `Cycles` is even. The predefined reduction referenced in the returns part (`sum`) is described in Chapter 9.

```
for i in [1..Cycles/2] do
    val := 1.0/real(4*i-3) - 1.0/real(4*i-1);
returns sum( val )
end for * 4.0
```

This expression computes the sum of all calculated `val`'s and multiplies the result by `4.0`. Note that all the `val`'s can be computed independently. That is, the distributed control establishes `Cycles/2` instances of the body, and all the instances can safely execute in parallel. Also note that in the absence of iterative control, the number of body invocations can be determined by simply examining the distributed control parts and the values it references (values that are invariant to the **for** expression). Finally, use of the "real" type conversion is unnecessary in the denominators as the numerators are real.

6.1 Control

A **for** expression must have either body distribution or iteration control, and can have both. Body distribution control is also referred to as an in-expression list. It can introduce value names of type integer whose values are taken from arithmetic progressions, or value names whose values represent the scattered components of arrays or streams of any type.

Iteration control, which takes the form of a boolean expression preceded by reserved word **while** or **until**, can appear just before the body or right after the body. If the control appears before the body, then it executes before each execution of the body. If the control appears after the body, then it executes after each body execution. When the boolean expression in a **while** evaluates to **false**, or **true** in an **until**, the iteration stops.

It is illegal to have both pre-test and post-test iterative control in the same **for** expression, since they can be combined easily. However, as mentioned earlier, it is possible for both distribution and iteration control to appear within the same **for** expression. Then the body is executed iteratively, and the number of iterations is guaranteed not to exceed the number of values defined by the distribution control, and guaranteed to be no more than the number of iterations allowed by the iteration control.

6.1.1 Distribution Control

In this section we describe the distribution control part. Remember that this part is optional, but if present, it must immediately follow the **for** reserved word. It introduces one or more named sets of integers taken from arithmetic progressions (called range triplets) or elements taken from arbitrarily typed array or stream values, or both. The scope of the names includes all other distribution control parts textually following it (if part of a **cross** product, which is discussed in Section 6.1.1), all iteration control parts, all expressions in the body, and all expressions in the results part, except when redefined in an inner scope. Names defined this way may not be redefined via assignment in the body of a **for**. The syntax follows:

in-exp-list	::=	in-exp [**dot** in-exp] ···
	|	in-exp [**cross** in-exp] ···
in-exp	::=	value-id **in** expression [**at** `[` index-id-list `]`]
	|	value-id **in** `[` triplet `]`
index-id-list	::=	{ .. | value-id } [, { .. | value-id }] ···

Range Triplets

A range triplet defines a named sequence of integer values taken from a triplet defining an arithmetic progression. The triplet must be enclosed in square brackets, and its expressions (each of which is optional) must be of type integer. The first two expressions of the triplet give the lower and upper bounds of the progression. The third expression gives the stride of the progression and defines its direction; the stride can never be zero (see Chapter 11). If the first expression of the triplet is missing, it defaults to 1. If the second expression is missing, it defaults to positive or negative infinity depending on the sign of the stride. The default stride is always 1. Consider the following examples:

```
I in [1..3]
J in [ ..3]
K in [3..100..2]
L in [100..98..-1]
M in [1..]
N in [..]
```

The first and second triplets are identical, as are the last two which are infinite progressions of stride 1. The value name (also called a triplet index) of a range triplet takes on the values of the arithmetic progression it is associated with. For example, `I` above successively takes on values 1, 2, and 3, while `L` successively takes on values 100, 99, and 98.

The following **for** expression uses a range triplet to distribute the integers between `1` and `N` and sum them:

```
for I in [1..N] do returns sum(I) end for
```

If `N` is `3`, then the expression returns `1+2+3`, or `6`. Notice that the body is empty.

Stream Scattering

The distribution control of a **for** expression can scatter stream components. The syntax is given by the first production of:

in-exp	::=	value-id **in** expression [**at** `[` index-id-list `]`]
	|	value-id **in** `[` triplet `]`

The expression following reserved word **in** must be a stream. The optional **at** part is simply a value name, which, for each stream element, is the integer position of the element in the stream.

For an example, consider

```
for x in S do returns sum(x) end for
```

which sums the elements of stream `S`.

Array Scattering

The distributed control of a **for** expression can also scatter array components using the same syntax as that for stream scattering. Here the expression following reserved word **in** must be an array. If an **at** part is provided, a value-id or empty triplet must be provided for each dimension of the scattered array. The **at** part gives the dimensionality of the scatter and introduces additional value names, called scatter indices, that take on the index positions of the scattered values. In the absence of an **at** part, all dimensions of the array are scattered; that is, the element index takes on all the components of the array. The empty triplet version specifies that the entire associated dimension is to be scattered.

The following expression scatters planes taken from the second and third dimensions of `A` (a three dimensional matrix):

```
for X in A at [I,..,..] do
   . . . X[..,..] . . .
returns . . .
end for
```

The dimensionality of `X` in this example is two, and `I` takes on the successive indices of `A`'s first dimension. The following example scatters all the components of `A`:

```
for X in A do  % identical to "for X in A at [I,J,K] do"
   . . . X . . .
returns . . .
end for
```

DOT and CROSS Products

The distribution control of a **for** expression can comprise one or more in-expressions via an in-expression list:

in-exp-list	::=	in-exp [**dot** in-exp] $\cdots$
	|	in-exp [**cross** in-exp] $\cdots$

Two or more in-expressions can be separated by reserved word **dot** to produce a dot product of in-expressions, all of which must yield the same number of values.

For example,

```
for i in [1..2] dot j in [3..4] do
returns product(i+j)
end for
```

Here the dot product produces two index pairs, `[1,3]` and `[2,4]`, and the entire expression yields `24`. See Chapter 9 for discussion of predefined reductions such as `product` above.

Two or more in-expressions can be separated by reserved word **cross** to yield a Cartesian, or outer product. Differing from **dot** in-expressions, the in-expressions involved in a **cross** need not produce the same number of values.

For example,

```
for i in [1..2] cross j in [3..4] do
returns product(i+j)
end for
```

which is structurally similar to the previous example, but with **cross** replacing **dot**. Here the cross product produces four pairs, `[1,3]`, `[1,4]`, `[2,3]`, `[2,4]`, and the entire expression yields `600`.

It is possible for triplet indices and element indices to be referenced in textually succeeding in-expressions of cross products. For example, the following expression yields `14400`:

```
for i in [1..2] cross j in [i..4] do
returns product(i+j)
end for
```

Notice that `i` is referenced in `j`'s triplet.

6.1.2 Iteration Control and Carried Values

Here is an example of an iterative **for** expression:

```
for
  I := 1
while I < 5 do
  K := I;
  I := I + 2;
  J := K + I;
returns product(I+J)
end for
```

The expression yields `91`, the product of `(1+2)+(1+(1+2))` and `(3+2)+(3+(3+2))`. Each carried value (here `I`) is redefined exactly once in the body. The semantics of conventional sequential execution describes the operation of each body.

Note that the initial value of a carried value need not be defined in the top portion of the **for** expression. The following example is legal in SISAL and the **for** expression within the **let** expression is semantically equivalent to the previous example:

```
let
    I := 1
in
    while I < 5 do
      K := I;
      I := I + 2;
      J := K + I;
    returns product(I+J)
    end while, I          % NOTE THE REFERENCE TO I WHICH IS THE
end let                   % I DEFINED IN THE LET PART
```

This expression yields `91, 1`. Notice that the **for** expression ends with reserved word **while** and not **for**.

In general, a carried value may be explicitly initialized in the top part of a **for** expression, or may be implicitly initialized merely by its use. In the latter case, the value with this name is copied for use within the **for**; the exterior name is not changed by the operation of the **for** (see the example above). This flexibility allows names which happen to be already defined (formal parameters, for example) to be used iteratively without spurious redefinition, and allows needed initializations to be done conveniently.

6.2 Result Values

The bottom or **returns** part of the **for** expression executes once for each execution of the body. It packages the results of the expression and returns them to the enclosing scope. The syntax of the returns part follows:

for-bottom	::=	**returns** return-exp [, return-exp] ···
		end { **for** | **while** | **until** | **do** }
return-exp	::=	**stream of** expression [filter]
	|	expression [filter]
	|	**suffix** `(` value-id `)`
	|	**array** [`[` [size-descriptor] `]`] **of** expression
		[filter | **at** `[` at-exp [, at-exp] ··· `]`]
size-descriptor	::=	[value-id **in**] triplet [, [value-id **in**] triplet] ···
filter	::=	**when** expression
	|	**unless** expression
at-exp	::=	expression | `..`

A packaging part can build either a stream (the **stream of** and **suffix** parts) or an array (the **array of** part), reduce a sequence of values into a single value (by applying a predefined reduction operation to the sequence), or eliminate all but the last value of a sequence and return that value (the expression part, which is also referred to as the "last value" part). We discuss each in the remaining sections, as well as the masking filters that can remove values from the various sequences. Note that the order of values in each sequence is identical to the body execution order, which is a function of the **for** expression's control logic. Semantically, the order of body executions for a distributed **for** expression is the same as a left-to-right progression across the controlling triplet (first expression to last expression in each triplet); controlling array scatter (first dimension to last dimension considering each dimension in subscript order), or controlling stream scatter (first component to the last component). For **cross** in-expressions, semantically the body execution order is from left to right; that is, from the first in-expression to the last in-expression.

If the body does not execute, then the returned values are those defined by the default actions of the various packaging parts. Note also that initial values for names of carried values do not contribute to the results of a **for** expression. As each body executes, new values are defined, and these names may be used to define results.

6.2.1 Last Value

The expression or "last value" part yields the last value in a sequence or reduces the sequence into a single value by applying a predefined reduction. The default value of this packaging part is either an error value or the default value of the applied reduction (refer to Chapter 9).

The following example illustrates both forms of the "last value" parts, and yields `19, 9, 100` as its results:

```
for I in [1..10] do
  J := I - 1;
returns I+9, J, sum(I+J)
end for
```

That is, it yields the last value of `I` plus `9`, the last value of `J`, and the sum of all the `(I+J)`s.

6.2.2 Stream of

The `stream of` part packages a sequence of values into a stream. The ordering of the packaged components is identical to the ordering of the sequence. For example, the following **for** expression returns a stream comprising integers 2, 3, and 4, respectively. That is, `S[2]` yields 3.

```
S := for i in [1..2] cross j in [i..2] do
     returns stream of i+j
     end for;
```

Note that the default value of the `stream of` part is an empty stream.

Using the **suffix** operation it is possible to return that portion of a stream that was not scattered by an in-expression; that is, a stream comprising those stream components that did not contribute to an execution of the body. This operation cannot appear within a **for** expression that has **cross** in-expressions. The only parameter to this operation may be the name of a stream being scattered by an in-expression. An example of its use follows, here we assume `S` is a stream of characters:

```
for c in S
while ( c = ' ' ) do
returns sum(1), suffix(S)
end for
```

The expression returns the leading number of blanks in `S` and a stream comprising the elements of `S` minus the leading blanks. If all the components of a scattered stream contribute to the various body executions, then the suffix stream is empty. The value returned if the body never executes is the original stream. Remember that the position of the first extant component of any stream is 1.

6.2.3 Array of

The `array of` part packages a sequence of values into an array. In the absence of an **at** part, which functions as a position specifier, the ordering of the packaged components is identical to the ordering of the sequence. For example, the following **for** expression returns a one-dimensional array comprising integers `2`, `3`, and `4`. That is, `A[2]` yields `3`.

```
A := for i in [1..2] cross j in [i..2] do
     returns array of i+j
     end for;
```

Here the lower bound of `A` is `1` (the default).

Providing a size descriptor in the `array of` part allows the packaging of multidimensional arrays. Distributed names or carried values may not be involved in the expressions giving array size. In this descriptor, the default lower bound of each triplet is 1, the upper bound may be omitted, and the stride is always 1 (the third expression of each triplet is never allowed), and named triplets are never allowed in this context. If an upper bound is given, then the extent is mandated, and the same number of values must contribute to filling the array elements. If one upper bound is supplied, they must be supplied for all dimensions. For an example, consider

```
let
    A := array integer [1..2,0..1: [1,..] 1, 2; [2,..] 3, 4 ];
    B := for I in [1..4] do
         returns array [1..2,0..1] of I
         end for
in
    A, B
end let
```

which builds two identical matrices. Notice that the values are placed in the resulting array in subscript order (left to right and top to bottom); that is in positions `[1,0]`, `[1,1]`, `[2,0]`, and `[2,1]`.

In a **for** expression using iteration control and carried values, all contributing expression values must have dimensionality one smaller than the descriptor and be the same size in all dimensions. In the array result, the extent of the first dimension comes from the number of iterations of the **for**, and the extent of the remaining dimensions comes from the common extents of the contributing expressions. If a one-dimensional array is being constructed, the contributing values may have any type; if the contributing values are arrays, they must have the same dimensionality, but may have different extents. That is, "ragged" arrays of arrays may be built in this way.

In a **for** expression using only distribution control, the dimensionality of the Cartesian product of values scattered via **cross** and **at** need not match the dimensionality of the array descriptor of the value being built.

- The descriptor may be absent or define a one-dimensional result, regardless of the scattering at the top of the **for**. For example,

  ```
  for i in [1..2] cross j in [3..4]
  returns array of i*j
  end for
  ```

  builds the same array as the generator

  ```
  array integer[ 3, 4, 6, 8 ]
  ```

  In this case, the contributing values may be of any type; if arrays, they must have the same dimensionality but may differ in extent (ragged arrays).

- The descriptor may specify a multidimensional result that matches the number of scattered items. In this case each corresponds to an associated array dimension, defining its extent. The contributing values can be of any type, each defining a single element. For example,

  ```
  for x in A at [i,j]
     k := . . .
  returns array[0..,0..] of g(x,k)
  end for
  ```

  defines a two-dimensional array whose extents in each dimension are defined by the number of `i` and `j` values, respectively.

- The descriptor may be n-dimensional in a **for** expression having k dimensions of distribution scatter, with $n > k$. Then the first k dimensions of the result are defined by the scatter extents. The contributing values must be $(n-k)$-dimensional with identical extents. They define the extents of the last $n-k$ dimensions. In the following, `g(x,k)` must produce two-dimensional arrays of identical extents to fill out the last two dimensions of the array result.

  ```
  for x in A at [i,j]
     k := . . .
  returns array[0..,0..,1..,1..] of g(x,k)
  end for
  ```

The optional **at** part is semantically similar to the position specifiers of an array generator (again refer to Chapter 4). That is, it specifies where individual components of the array should be placed. Like an array generator, one and only one component can be defined for each component position in the array, and the presence of an **at** part requires the presence of a size descriptor that specifies the extents of all dimensions (upper bounds are not optional here). Here is a **for** expression that builds an array comprising components 40, 30, 20, and 10, respectively:

```
T := for I in [1..4] do
     returns array [4..7] of I*10 at [4+I-1]
     end for
```

Note that the lower bound of the resulting array is 4. The next example places the components of the vectors built within the **for** body into successive rows of the resulting matrix, called `M`:

```
M := for I in [1..2] do
          V := for J in [1..2] do
               returns array of I + J
               end for;
     returns array [1..2,1..2] of V at [I,..]
     end for;
```

Note that the number of parts of the position specifier must match the dimensionality of the generated array. For a complete discussion of position specifiers and the semantics of array generation, refer to Chapter 4.

In the absence of a size descriptor, the default value of an `array of` part itself is an empty array if the **for** body never executes. If a size descriptor is present, the default value is an error value (see Chapter 11).

6.2.4 Masking Filters

Except for the **suffix** operator, each returns part can be optionally followed by a masking filter. In array packaging, a masking filter is not allowed if an **at** part or size descriptor is present, and makes sense only for one-dimensional arrays (to preserve rectangularity of multidimensional arrays). A masking filter is a boolean expression preceded by either a **when** or **unless** reserved word. Its syntax follows:

filter	::=	**when** expression
	|	**unless** expression

Both forms determine if specific sequence values should be disregarded during packaging. If a **when** filter is **false** or if an **unless** filter is **true**, the corresponding sequence value is dropped from the sequence. The following example generates a one dimensional array of odd integers starting with 1:

```
for I in [1..N] do
returns array of I when mod(I,2) ~= 0
end for
```

Chapter 7

Records and Unions

This chapter describes SISAL's record and union types, and their respective operations.

7.1 Records

A record is a collection of one or more values, of possibly different type. This is in contrast to an array or stream, where the components must be of the same type.

7.1.1 Record Type Specification and Conformity

The syntax for specifying a record type is:

record-spec	::=	**record** [field-spec $\big[$; field-spec $\big]\cdots\big[$;$\big]$]
field-spec	::=	field-id $\big[$, field-id $\big]\cdots$: type-spec
field-id	::=	name

For example, if `t1,...,tk` are SISAL type specifications and `n1,...,nk` are distinct names (called record fields), then

```
type T = record[ n1:t1; ...; nk:tk ];
```

is a record type called `T`. A simple record containing a single real value has type

```
type Example0 = record[ C: real ];
```

The type name of a record cannot be referenced as a type name within its own specification. That is,

```
type Stack = record[ Value:real; Rest:Stack ];
```

is illegal.

The name of a record type can be used as a field name within its specification. Further, a field name can be re-used within another record specification, and can be the same as another type name without conflict. For example, type declarations

```
Type Example1 = record[ Example1: record Example0[ Example1:real ] ];
```

and

```
Type Example2 = record[ Example1:Example1 ];
```

do not conflict. The field name of both records is `Example1`. The field of the first record is itself a record with one field of type real. The field of the second record is of type `Example1`.

Two record specifications conform if they have the same number of fields and the corresponding fields (in order) conform. Field names are not considered in record type conformance. For example, `XYrec` and `YXrec` shown below, do not conform:

```
type XYrec = record[ X:double;  Y:integer ];
type YXrec = record[ X:integer; Y:double ];
```

However,

```
type ABrec = record[ A:double;  B:integer ];
```

conforms with `XYrec` even though corresponding field names differ.

7.1.2 Record Generation and Replacement

SISAL provides a single operation for building record values and a single operation for replacing record fields; the syntax is:

record-gen	::=	**record** type-spec `[` field-def $\big[$; field-def $\big]\cdots\big[$;$\big]$ `]`
	|	primary `[` field-def $\big[$; field-def $\big]\cdots\big[$;$\big]$ `]`
field-def	::=	field-id $\big[$. field-id $\big]\cdots$: expression

The first rule of the record-gen production describes the construction operation. All the field names found in the type specification must receive values exactly once, and the constructed record must conform to the type specification. Consider

```
record XYrec[ X:2.0D0; Y:1 ]
```

which builds a record of type `XYrec` (defined in the previous section). Note that this is equivalent to `record XYrec [ Y: 1; X:2.0D0 ]`, because the given type specification provides field names and order. The fields in the record generator thus may appear in any order.

The second rule of the record-gen production describes the record replacement operation. No component may be mentioned twice. Its result is a record similar to the leading expression except having one or more different field values. For example, assuming `Rec` names a value of type `XYrec`,

```
Rec[ X:1.0D0 ]
```

yields a record identical to `Rec`, except with field `X` changed to `1.0D0`.

Note that abbreviated notation is provided for compound replacement operations. For example, if `ExRec` names a value of type `Example1` (defined in the previous section), then

```
ExRec[ Example1.Example1: A+2.3 ]
```

is a new compound record whose innermost scalar constituent is `A+2.3`. An equivalent replace operation that does not use the abbreviated notation is

```
ExRec[ Example1: record Example0[ Example1:A+2.3 ] ]
```

7.1.3 Record Field Selection

Dot notation selects the value of a record field. The syntax is:

record-ref	::=	primary . field-id

The value of the leading expression must be a record having the named field. Because the record reference operation is itself an expression, a value name followed by a sequence of field names separated by dots can select innermost values of compound records. For example, assuming A is of type `Example2` (defined in Section 7.1.1), `A.Example1` yields the component record, while `A.Example1.Example1` yields the scalar of the component record.

7.2 Unions

A union is a collection of one or more possible values, each possibly of different type. That is, a union value can have only one element taken from a collection of several constituent types. Except for tag test operations, the union operations are a syntactic subset of the record operations.

7.2.1 Union Type Specification and Conformity

The syntax of a union's type specification is similar to the syntax of a record's type specification. The exceptions are:

- Replace reserved word **record** by **union**.
- A union's type name can appear as a type name within its specification, as long as the union has at least one component that does not use the union's type name.

Also, for clarity, the constituents of a union type are called tags and not fields, as they are for the record types.

The rules governing conformity of two union types are identical to those for records (refer to Section 7.1.1).

Syntax:

union-spec	::=	**union** [tag-spec $\big[$; tag-spec $\big]\cdots\big[$;$\big]$]
tag-spec	::=	tag-id $\big[$, tag-id $\big]\cdots$: type-spec
tag-id	::=	name

Example:

```
type UnEx1 = union[ T1:real; T2:double ];
type UnEx2 = union[ T1:integer; T2:UnEx1 ];
```

7.2.2 Union Generation and Replacement

SISAL provides a single operation for building a union value and a single operation for replacing a union's element; the syntax is:

union-gen	::=	**union** type-spec [tag-id : expression $\big[$;$\big]$]
	|	primary [tag-def $\big[$;$\big]$]
tag-def	::=	tag-id $\big[$. tag-id $\big]\cdots$: expression

The first rule of the union-gen production describes the construction operation. Only one tag associated with the union under construction can receive a value. We say that this is the union's *active* tag. Consider the following example that builds a union of type `UnEx2` (defined in the previous section):

```
union UnEx2 [ T1:1 ]
```

The second rule of the union-gen production describes the union replacement operation. It yields a union similar to the leading expression except having a new element (of the type associated with the specified tag). For example, assuming `Un` names a value of type `UnEx2`,

```
Un[ T2: union UnEx1 [ T1:2.0 ] ]
```

yields a new union with `T2` activated.

Note that abbreviated notation is provided for compound union replacement operations, as it is for record replacement operations.

7.2.3 Union Field Selection

Dot notation selects the value of a named tag. The syntax is:

union-ref	::=	primary . tag-id

The value of the leading expression must be a union having the named tag. Because the union reference operation is itself an expression, a value name followed by a sequence of tag names separated by dots selects innermost values of a compound union.

7.2.4 The NULL type

The null type occurs most often within union types where data values are not required, such as in the definition of enumeration types or data types of unlimited depth. The proper element of the null type is **nil**. SISAL's arithmetic operations do not work with data of type null.

Here is a data type of unlimited depth:

```
type StNode = union[ Empty:null;
                           Element:record[ Value:real; Next:StNode ] ];
```

which defines a list.

7.2.5 Tag Test

The tag test operation checks if a specific tag of a union is active, yielding **true** if it is, else **false**.

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

The expression must be a union, and the tag-id must specify a union tag. For example, this expression yields **false**:

```
let
   UnValue := union UnType [ red:nil ];
in
   is green( UnValue )
end let
```

assuming

```
Type UnType = union[ red, green, blue, black, white : null ];
```

This example also illustrates the use of unions to build enumerations.

Chapter 8

Selection Expressions

In this chapter, we present SISAL's selection constructs: **if** and **case**. They are themselves expressions, and may appear in arithmetic expressions if they have arity one.

8.1 IF Construct

The **if** construct selects one expression list, depending on the values of one or more boolean expressions. An alternative version of the **if** construct allows a limited form of type inquiry.

Syntax:

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

The expressions following the **if** and **elseif** reserved words are test expressions. They must be of boolean type. The expression lists following the **then** and **else** reserved words give the arms of the construct. All the arms must have the same arity and type.

The values produced by the **if** construct are those produced by the lexically arranged first arm whose test expression is **true**, or the final arm if all the test expressions are **false**. The selection process begins with the **if** and proceeds through the **elseif**s in order of presentation.

The **if** construct does not introduce value names. However, all value names defined in the scopes in which an **if** is embedded are accessible within it.

Example:

```
if P < 10.0 then
```

```
    Y+4.0, 0
elseif P = 10.0 then
    Y, 1
else
    Y**2.0, 2
end if
```

This expression has arity two. The first expression of all three arms yields a value of type real, and the second yields a value of type integer. The **if** yields either `Y+4.0` and `0`, `Y` and `1`, or `Y**2` and `2`.

In an alternative form, type values may be compared in an **if** construct. In this case the restriction that all arms must have the same type is relaxed. For example,

```
if type(z) = real then 0.0 else 0 end if
```

has as value a real or integer zero, depending on the type of `z`. This kind of usage is primarily associated with accessing values whose types are defined via typesets (see Chapter 9).

8.2 CASE Construct

The case expression selects an arm based on the value of a controlling expression. The controlling expression, however, need not be of boolean type. Thus, the **case** construct is a generalization of the **if** construct. An alternate form of the **case** construct allows a limited form of type inquiry. Its syntax follows:

case-exp	::=	**case** expression **of**
		case-part [; case-part] $\cdots$
		[; **otherwise** : expression-list] [;]
		end case
case-part	::=	pattern [, pattern] $\cdots$: expression-list
pattern	::=	**false**
	|	**true**
	|	integer-constant
	|	character-constant
	|	character-string-constant
	|	tag-id
	|	type-spec

The expression following the **case** reserved word controls the selection. It must be of either boolean, integer, character, array of character, or union type. If the controlling expression is a union value, then each provided pattern must be a union tag. If the type of the controlling expression is an array of character or a basic type, then each provided pattern must be a constant of that type. All the patterns must be distinct; however, one or more patterns can be associated with the same arm. The **otherwise** pattern represents all patterns not explicitly stated in the construct; it is required if all possible patterns have not been listed to exhaust the values of the type. It must not appear if all possible values have been used.

The arm associated with the pattern matching the value of the controlling expression yields the results of the **case**. If the control fails to match a provided pattern, the arm associated with **otherwise** is selected. If the the control is of type union, then the pattern matching the active tag of the control is selected. Again, if the tag of the control is not provided as a pattern, the arm associated with **otherwise** is selected. If the control is an array of characters (or string), then the arm having a string pattern whose constituents are of equal kind and number will be selected—the lower bound of the control is not relevant.

As with the **if** construct, the arms of the **case** construct must conform in arity and type, and the entire **case** must conform to its arms. Also, as with the **if** construct, the **case** construct cannot introduce value names. However, all value names defined in the scopes in which a **case** is embedded are accessible within it.

Example:

```
case word of
  "data",
  "flow"    : "Yes!";
  "hello"   : "Good-Bye";
  otherwise : "Don't Care";
end case
```

If the value of `word` is `"hello"`, then the expression will yield `"Good-Bye"`. If the value of `word` is either `"data"` or `"flow"`, then the expression will yield `"Yes!"`. Note that for this example, the **otherwise** reserved word is required because all possible values of the type cannot be listed.

For another example, consider:

```
case node of
  link: Traverse(node.link);
  data: node.data;
  tail: 0;
end case
```

where `node` is of type

```
type NodeType = union[ tail:null; link:NodeType; data:integer ];
```

This example expression yields the value from function `Traverse`, if `link` is `node`'s active tag. If `data` is the active tag, then the expression will yield `node.data` as its result. Failing both, the result is `0`. Note that the **otherwise** reserved word, unlike the previous example, is illegal—all tags of `NodeType` are provided as patterns.

Example:

```
case D of
  10, 20, 30 : 10, D+1;
  40, 50     : 40, D+2;
  otherwise: :  D, D+3;
end case
```

This expression has arity two. If the value of `D` is 10, 20, or 30, then the expression will yield 10 and `D+1` as its results. If `D` is 40 or 50, then the expression yields 40 and `D+2`. If the value of `D` does not match any of the five patterns, however, than the expression will yield `D` and `D+3`. Note that, as in the first example, the **otherwise** reserved word is required.

In the alternate form, a type value may be used in a **case** construct. Then each pattern is a type specification; as in the **if** construct, the arms are not restricted to being of the same type. For example:

```
case type(z) of
  integer: 0;
  real   : 0.0;
  double : 0.0d0;
  complex: (0.0, 0.0);
  double_complex: (0.0d0, 0.0d0)
end case
```

Using type values is useful in conjunction with values whose types are defined with typesets (see Chapter 9).

Chapter 9

Functions, Reductions, and Typesets

This chapter describes SISAL's function definitions and invocations, reduction invocations, and typesets. The syntax of the constructs discussed in this chapter is shown below:

function-spec	::=	**function** ([type-spec-list] **returns** type-spec-list)
type-spec-list	::=	type-spec [, type-spec] ···
function-header	::=	function-id ([param-decls] **returns** type-spec-list)
param-decls	::=	param-decl [, param-decl] ··· [,]
param-decl	::=	value-id [, value-id] ··· : type-spec
function-def	::=	**forward function** function-id ([type-spec-list]
		returns type-spec-list) [;]
	|	**function** function-header
		expression-list
		end function
invocation	::=	primary ([[expression] [, [expression]] ···])
basic-type	::=	**boolean**
	|	**character**
	|	**integer**
	|	**real**
	|	**double**
	|	**complex**
	|	**double_complex**
	|	**null**
	|	**foreign** language-id **function**
function-id	::=	name
reduction-id	::=	name

Of the basic types, the first eight were discussed in Chapter 3, as were the the type conversion prefix operations.

9.1 Function Definitions and Invocations

A function definition comprises a function header followed by an expression list that defines the function's body.

A function header declares the name and type of each formal parameter to the function (between the left parenthesis and reserved word **returns**). The declarations are separated by commas, but each declaration may contain several value names, which are themselves separated by commas. The scope of a formal parameter is the function's body, less any inner constructs that reintroduce the same value name. The type of each value returned from a function (there must be at least one) must be defined by the type specification list following the **returns** reserved word. This type list must conform to the expression list composing the function's body.

The number of actual parameters passed to a function must match the number specified in its function header, and their respective types must conform. We exemplify the parameter passing mechanism below. Given

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

invocation

```
F( parameter-expression-list );
```

is equivalent to

```
let
    A1:T1, ..., An:Tn := parameter-expression-list
in
    <F-body>
end let
```

However, the scoping rules for functions and **let** expressions are not the same. This is the subject of the next section.

9.2 Inheritance of Definitions and Declarations

A function has access only to the data presented to it via its parameters. Nested function definitions are not allowed.

Type declarations and function definitions made before a function's definition and within an enclosing scope are automatically inherited by the function, but can be reintroduced within the function.

9.3 Forward Declarations

The use of mutual recursion requires that the name and parameter and result types of a function be available before the definition of that function. The **forward** declaration provides this information. For example,

```
forward function Goo( double returns double );

function Foo( C:double returns double )
    if ( C > 1000.0 )  then 1.0 else Goo( C + 1.0 ) end if
end function

function Goo( V:double returns double )
    Foo( V )
end function
```

The forward function declaration allows `Goo` to be referenced in function `Foo`, which is defined before `Goo`. The function header of the forward declaration and the function header in the function's definition must conform, although the formal parameter names do not appear in the forward declaration.

9.4 Function Types

In SISAL, a function is a first class object. As such, a function type declaration yields a legal compound type, and can appear anywhere a compound type can appear. For example, `FuncType`, shown below, is a function type that represents a function taking two parameters (a real and an integer) and returning two values (a real and an integer):

```
type FuncType = function ( real, integer returns real, integer );
```

Given this type definition, the following type represents an array of functions:

```
type ArrType = array of FuncType;
```

and the following expression, assuming `A` is such an array, illustrates how such a function might be invoked:

```
B, C := A[TheOneIWant]( -3.14235, 2010 );
```

A function type might not describe a SISAL function, but a foreign function. Such a function type is declared by following reserved word **foreign** with the name of the language in which the function is defined—this is followed by reserved word **function** to complete the declaration. For example, `FFuncType` shown below represents a foreign FORTRAN function:

```
type FFuncType = foreign FORTRAN function;
```

Note that this type can only be used to declare formal parameters of other foreign functions (refer to Chapter 10).

Two function types conform if their parameter lists and returns lists conform in type.

9.5 Function Definition by Assignment

Function definitions are necessary for the starting points of execution in program units and as public functions in module units. In addition, a function can be defined via an assignment within a **let** construct or a **for** expression—in any context where a value of any type can be defined. Function assignment was introduced but not discussed in Chapter 3. The function name being defined is followed by a complete function header. The parameter names in the header can be referenced within the expression defining the function's body. Unlike functions defined as stand-alone units, the expression list for the function body may refer to value names from the context of assignment, which are fixed as part of the function definition. For example, consider

```
let
   s := 10.0;
   Increase(x: real returns real) := x * s
in
   for elt in A do
   returns array of Increase(elt)
   end for
end let
```

Function `Increase` multiplies its parameter by `10.0` because the value of `s` is bound to the definition of `Increase`.

The following example shows a more complicated usage of functions defined by assignment:

```
for
   i := 1;
   f(x: real returns real) := x + x*x
while i <= n repeat
   i := i + 1;
   f(x: real returns real) := f(x) * x
returns f
end for
```

This expression has type `function(real returns real)` and its value is a function whose defining expression is equivalent to `x*(x*( ...  (x + x**2) ...  ))`.

9.6 Higher Order Functions

Functions can be passed to and returned from functions, and be the values of expressions. We discuss each in turn.

9.6.1 Functions as Parameters

Functions may have function types as parameters, just as they can take other SISAL types. For example, function `Integrate` below takes a function (with a real parameter and returning a real result) as its first parameter:

```
type FT1 = function( real returns real );

function Integrate( F: FT1, A, B: real returns real )
    . . .
end function
```

Invoking function `F` within the body of `Integrate` is like calling any other function.

9.6.2 Functions as Values

The result of an expression can be a function. The function below decides which of two functions should be evaluated:

```
function Decide( F1, F2: FT1, B: boolean returns FT1 )
    if B then F1 else F2 end if
end function
```

and the following function returns an array of `FT1` using function `Decide`:

```
function EG( F1, F2: FT1, A: array of boolean returns array of FT1 )
   array[i in liml(A)..limh(A): Decide(F1, F2, A[i]) ]
end function
```

Note that a type declaration for function `EG` above is

```
type FT2 = function (FT1, FT1, array of boolean returns array of FT1);
```

9.7 Reduction Invocations

A reduction is a SISAL element that combines a sequence of values into a single value. It can be invoked in the "last value" version of a **returns** part in a **for** expression, or can be used in an arbitrary expression like an intrinsic function when applied to arrays and streams. SISAL provides several predefined reductions. Although a reduction resembles a function in application, one should not confuse the two ideas. A reduction always incorporates a series of values into a final result which is available after the last item has been incorporated.

minval	maxval	catenate	sum	product	histo

minval, maxval: Yield the least and greatest value of a sequence of integer, real, or double values. Applying these reductions to an empty sequence of either integers, reals, or doubles yields the largest or smallest value for that type on the host machine.

sum: Returns the sum of a sequence of integer, real, double, complex, or double_complex values. Applying this reduction to an empty sequence results in `0`, `0.0`, `0.0D0`, `(0.0,0.0)`, or `(0.0D0,0.0D0)` respectively. When applied to a sequence of boolean values, it yields the logical disjunction of them; an empty sequence yields **false**.

product: Returns the product of a sequence of integer, real, double, complex, or double_complex values. Applying this reduction to an empty sequence results in `1`, `1.0`, `1.0D0`, `(1.0,0.0)`, or `(1.0D0,0.0D0)` respectively. When applied to a sequence of boolean values, it yields the logical conjunction of them; an empty sequence yields **true**.

catenate: Returns the concatenation of a sequence of one-dimensional arrays or a sequence of streams. The resulting value is a single array or stream whose type matches the parameter type. Applying this reduction to an empty sequence results in an empty array or stream. The lower bound of a resulting array is always 1. For example, the following expression returns an array of integers comprising 1, 2, 3, 4, and 5:

```
let
  A := array integer [ 1, 2, 3 ];
  B := array integer [];
  C := array integer [ 4, 5 ]
in
    catenate( A, B, C )
end let
```

histo: This reduction is somewhat different from the others. Its value is a histogram array built by applying one of the other predefined reductions to a set of values. The reduction accepts five parameters:

```
histo( reduction_id, low, high, value, index )
```

- `reduction_id`: The name of one of the other predefined reductions; this is the operation performed at each histogram reduction step. This choice also implicitly specifies the initial values for elements of the result array.
- `low`: The lower bound of the result array.
- `high`: The upper bound of the result array. If it is less than the lower bound, an empty array results.
- `value`: The value which is combined by the reduction with result array elements.
- `index`: The position in the result array where the reduction will combine the value with the array element.

The histogram reduction can appear in two contexts, like the others. First, it may be used in the **returns** part of a **for** expression, as follows:

```
for ... do
   ...
   value := ... ;
   index := ... ;
   ...
returns histo( sum, 1, 100, value, index )
end for
```

Here the type of `value` determines the component type of the resulting array, and `index` is an integer. At each step the sum reduction is applied to the array element at `index`, and `value`. Because `sum` is specified, the histogram array is initialized to zeros.

Second, a histogram can be built by a stand-alone application:

```
let
   x := histo( sum, 1, 100, value, index )
   ...
```

In this instance `value` is an array or stream of values whose component type is the component type of the histogram array, and `index` is an array or stream of integers. Corresponding elements of `value` and `index` are used to apply `sum` repeatedly. One can use a scalar for `value`; it will be used with each `index` value.

9.8 Typesets

SISAL supports typesets to help generalize functions and the modules they compose. A typeset is a list of type specifications; the compiler infers which element is meant in a particular context.

type-set ::= type-spec [{ + | - } type-spec] ···

The operators + and - denote set union and difference, respectively. For example,

```
type alpha = integer + real + complex + boolean;
type arr_alpha = array [..] of alpha;
```

defines `alpha` to be a typeset representing one of the listed basic types, and `arr_alpha` to be a one-dimensional array of `alpha`. In their scope of definition, these types may be used like any other type unless operations on their values conflict. For example, it is a type error for operations + and ~ to reference a formal parameter of type `alpha` in the same function.

The following example defines a `map` function that applies a supplied function to an array of values, to produce another array:

```
function map( A: arr_alpha,
              f: function( alpha returns alpha )
              returns arr_alpha )
    for elt in A
    returns array of f(elt)
    end for
end function
```

Function `map` may be invoked with a first actual parameter that is an array of any of the types listed in the typeset `alpha`.

The repeated use of a type name defined as a typeset in a context does not mean that all occurrences must refer to the same type specification in the set. Each independently specifies that inference will be required to determine the type of each name. For example, the following fragment uses `map` to convert an array of integers to an array of complex numbers:

```
let
   I := array integer[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
   itoc( a: integer returns complex ) := ( real(a), 0.0 )
in
   map( I, itoc )
end let
```

Because they interact at the operating system level, no formal parameters to functions defined in a *program* can be defined by typesets. Neither can functions defined in a program return values of such types. Formal parameters and result types of functions declared in foreign definitions cannot be typed via typesets.

A few typesets are predefined in SISAL for convenience. We list them here. These definitions also illustrate the set union and difference operators for typesets.

type numeric	=	**integer + real + double + complex + double_complex**;
type lt_comparable	=	**character +** numeric **- complex - double_complex**;
type any	=	"Includes all types. The only operations allowed on values defined by this typeset are equality testing, assignment, and parameter passing."

Type inquiry and type values, discussed in Chapter 3, and type values in selection constructs, discussed in Chapter 8, are designed to work with values whose types are defined by typesets. In the following example, one needs a constant zero of an appropriate type to initialize an array:

```
function buildit( x: numeric, n: integer returns array of numeric )
   for a := array type(x) [ 1..n: case type(x)
                                     integer:        0;
                                     real:           0.0;
                                     double:         (0.0d0;
                                     complex:        (0.0, 0.0);
                                     double_complex: (0.0d0, 0.0d0)
                                  end case ]
   while condition(a) do
      a := improve(a)
   returns a
   end for
end function
```

In all cases a language processor can determine the appropriate selection before execution begins, and the result can be optimized to just that value.

Chapter 10

Programs, Interfaces, and Modules

10.1 Compilation Units

Compilation units are the texts that can be separately compiled by the SISAL compiler.

compilation-unit	::=	interface $\mid$ module
interface-id	::=	name
int-function-decl	::=	**function** function-id ([int-param-decls]
		[**returns** type-spec-list]) [;]
int-param-decls	::=	int-param-decl [, int-param-decl] $\cdots$
int-param-decl	::=	[**in** $\mid$ **out** $\mid$ **inout**] type-spec
language-id	::=	name
interface	::=	**interface** interface-id [{ **for** $\mid$ **in** } language-id]
		[import-decl $\mid$ type-def $\mid$ int-function-decl] $\cdots$
		end interface
module	::=	{ **program** $\mid$ **module** } module-id
		[import-decl $\mid$ type-def $\mid$ function-def] $\cdots$
		end { **program** $\mid$ **module** }
module-id	::=	name

The simplest form of compilation unit is a complete program that refers only to functions and types defined within it:

```
program X
    % complete set of type and function definitions
end program
```

A compilation unit defined using the reserved word **program** contains functions that can be called from the operating system level. Inputs to the program are the actual parameters of the called function, and the results of the called function are the outputs from the program. Types of parameters and results cannot be members of typesets here. Neither are function types allowed; compound values including typesets or function types are illegal. Opaque and state types are also illegal in this context.

A program may comprise more than one compilation unit, including one **program** containing functions that can start the computation. Certain functions and types defined in one unit may be used (imported) in others. In general, pairs of SISAL compilation units exist: one **module** that consist of a set of function and type definitions, and one **interface** that declares a subset of these functions and types to be usable by other units. The pairing is expressed by their both having the same name.

```
interface X
    % Declaration of public functions and types
end interface

module X
    % Includes a complete definition of the functions and
    % types in the corresponding interface, and possibly
    % some others
end module
```

The interface defines the relationship of the module with the world outside it. It contains public function headers and type declarations that can be referred to by other units. To prevent export of the named components of a type (record or union), the type declaration can be written incompletely in an interface—this is called an *opaque* type declaration. Opaque type declarations are not allowed in programs or modules. Here is an example:

```
interface B
   type r;
   function arf( returns r );
   function woo( x: r returns integer );
end interface

module B
   type r = record[ t: integer; u: real ];
   . . .
end module
```

An import declaration allows functions or types defined in one module, to be used in the compilation unit where the **from** appears.

import-decl	::=	**from** interface-id [: import-id [, import-id] · · ·] [;]
import-id	::=	type-id | function-id

If the list of imported identifiers is omitted, all public names from the interface are imported. Consider this example:

```
module Y
    from X: a, b, c;
    . . .
end module
```

When compiling `Y` above, some representation of the declarations for `a`, `b`, and `c` must be available. Hence the interface for `X` must have been written and compiled before compilation of `Y` can occur. However, the module for `X` need not exist at this time.

The scope of a name defined in an interface is the interface itself after the name definition, and all the units into which the name is imported; inner scopes in these places may redefine the name. Its scope does not include the associated module; here redefinition of each name in the interface is required. A type or function name defined in an interface and in a module must have conforming definitions. A name defined in a module has as its scope the entire module after the name definition, less any inner scopes that redefine the name.

Because two modules could both have a function with the same name, function names in a function call can be prefixed with a module-id: `t := X.b(1.0,2.0)`.

10.2 Interfaces for Foreign Languages

In general there is exactly one interface with a particular name and one module with the same name. There are two exceptions to this rule. First, a program has no corresponding interface. Second, an interface may declare the interface between a SISAL program and code written in another language.

We assume that the foreign language is imperative, that is, it relies on persistent state to operate.

10.2.1 Using SISAL from Foreign Code

A foreign language program can use functions from a SISAL module. Here there must be both a SISAL interface and module. The interface declares for which language an interface needs to be created, and declares which type and function definitions are accessible. The type definitions allowed in the interface are restricted to the intersection of the basic types available in the foreign language and in SISAL. For FORTRAN this means characters, integers, reals, doubles, and complexes[1], and rectangular arrays of these. For example:

[1]Double_complex is anticipated in the next FORTRAN standard.

```
interface FastParallelSort for FORTRAN
   type AInt = array of integer;
   function FPS ( AInt returns AInt);
end interface

module FastParallelSort
   type AInt = array of integer;
   function FPS ( A: AInt returns AInt)
       . . .
   end function
end module
```

Note the use of **for** to denote that the SISAL code is for use by a foreign language.

10.2.2 Using Foreign Code from SISAL

A SISAL program may use code written in a foreign language. Here only a SISAL interface is required since the foreign code stands for the nonexistent associated module.

Functional Foreign Code

In the simplest form the foreign code comprises well-behaved functions that have no state between calls and do not change their parameter values—just like SISAL functions. Then the declarations in the interface are the same as when interfacing with a SISAL module:

```
interface MathLib in FORTRAN
   function sin ( real returns real );
   function tan ( real returns real );
end interface
```

Note the use of **in** to denote that the code is written in a foreign language. Since functions in these languages return a single value, the **returns** part has only one type specification.

As another example, we may wish to define a compilation unit for the predefined functions of SISAL, not all of which are easily written in SISAL. Assume here that the compiler has mapped occurrences of the predefined function "size" to "size1D" for inquiries in the first dimension. Also, the compiler will invoke different versions for "abs".

```
interface Sisal_Predefined
    . . .
    function numabs( numeric returns numeric );
    function comabs( complex returns real );
```

```
        function size1D( array of any returns integer );
        . . .
    end interface

    module Sisal_Predefined
        . . .
        from Sisal_Clib: size_1D_C;

        function numabs( x: numeric returns numeric )
            if x < 0 then -x else x end if
        end function

        function comabs( z: complex returns real )
            (z.real**2 + z.imag**2)**0.5
        end function

        function size1D( x: array of any returns integer )
            size_1D_C( x )
        end function
        . . .
    end module
```

Foreign Code that Alters Parameters

A more complicated case is one in which the foreign code alters the values of parameters passed, and in which the code is written as proper procedures instead of value-returning functions. More information is needed in the interface to describe the behavior of the foreign code. Since proper procedures do not return values as functions do, the **returns** part in a function header is optional here. To capture the (re)assignment of values to function parameters as a side effect of function calls, a function header in an interface can include directions **in**, **out**, and **inout**, where **in** stands for a value parameter copied to the foreign function before function execution, **out** for a result parameter copied back to the SISAL call after function execution, and **inout** for a combination of the two.

The SISAL module containing this call must, of course, obey the single assignment rule. This implies that an **out** parameter will have no actual input parameter in the function call, but will assign a value to a SISAL value name, while an **inout** parameter copies the value of the actual parameter and assigns a value to a value name. To accomplish this, the arity of a foreign language function is the sum of the **out** and **inout** parameters, plus one if the **returns** part is present.

In the following example the result value of formal parameter `X` will be assigned to `t` and the result of formal parameter `Y` will be assigned to `a`.

```
    interface  VecSubs in FORTRAN
```

```
  function SUBA(inout array of real, in integer, in integer, out real)
end interface

module Y
    from VecSubs: SUBA;
    . . .
        let
            x := buildarray(1,100);
            t,a := SUBA(x,1,100,)
        in
     . . .
end module
```

Foreign Code with Persistent State

The most complex case is one in which the foreign code retains persistent state in static storage between references to its functions and procedures. In FORTRAN this would be in COMMON blocks and SAVE values. This possibility demonstrates the mismatch between the functional and imperative paradigms, and significant effort is required to overcome it. To approach a solution, we postulate the existence of a compiler system, for each such language, that alters the code in these ways:

- Each routine that accesses static storage is given a new first formal parameter that is a pointer to undifferentiated storage.
- Each reference to static storage is replaced by an expression that is the new first parameter plus an offset, which is then dereferenced to obtain the needed value or used to store information.
- Each reference to a foreign routine that accesses static storage must be augmented by a new first parameter that passes the pointer value along.

A compiler system that accomplishes the above storage reorganization and enforces these restrictions must also have access to the SISAL interface, whose structure is described below. Modifying an existing foreign language compiler is a modest task, and is justified if enough foreign language code of interest exists to make it more feasible than rewriting the code itself. New imperative code may be written to conform to these requirements.

A SISAL value with a special type state embodies all persistent data required by the functions in a foreign language module. The interface has two additional parts: an opaque type definition for this new type, and a function header for a function named **instance**, that takes no parameters, returning a value of type state. Foreign functions declared here that have persistent state have an **inout** parameter of type **state** as their first formal parameter. The state will be initialized by a function call to **instance** that returns an initial value representing the collection of all persistent state these routines collectively use. Then a SISAL program passes this value in to a foreign function and receives it back as a function result, and passes it back in again to the next foreign function reference. Note that this naturally imposes a dataflow ordering on the references to this foreign code.

```
interface M in FORTRAN
    type state; function instance( returns state );
    function SUBB(inout state, out real, in real)
end interface

module Y
    from M: state, instance, SUBB;
    . . .
       let
           s1 := M.instance(); s2, a := SUBB(s1, , 2.0)
       in . . .
       end let
    . . .
end module
```

Chapter 11

Errors

SISAL includes definitions of results from erroneous computations. An implementation may elect to halt all processing of a SISAL program when an error is detected. In this chapter we explain how error values arise and are propagated by an implementation that chooses to include error processing.

SISAL handles exceptional conditions by producing error values. These special values serve two purposes. First, they show that some kind of computational difficulty prevented production of a value. Second, they preserve portions of structured objects known to be correct; that is, those not affected by computational difficulties.

To this end, **error** is a proper element of every SISAL type (primitive or user-defined). Arithmetic events that could produce this value include overflow, underflow, and division by zero. Control flow errors that could produce this value include conditional expressions with test parts producing error values, and **for** expressions with test parts producing error values. SISAL also defines rules for propagating error values through expressions. In general, an error value cannot be canceled out of a calculation without explicit program testing.

11.1 Error Values

The full name of an error value consists of reserved word **error** followed by a type specification enclosed in brackets. Syntactically:

error-value	::=	**error** [type-spec]

The type specification is required as every value must have a known type. For example, **error**[**real**] differs from **error**[**integer**].

11.2 Testing for Error Values

The error test operation returns **true** if it has an error-valued input, else it returns **false**. Syntactically:

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

This operation must be used instead of equality to detect error values, as the boolean expression

```
X = error[real]
```

always yields an error value (refer to Section 11.3.1).

11.3 Type-Specific Operations Causing Errors

In this section we defined the error semantics of SISAL's type-specific operations. Unless specified otherwise, each type-specific operation yields an error value if it has one or more error-valued inputs.

11.3.1 Relational and Boolean Operations

Relational operators yield **error** if either operand is **error**.

The `&` operation always yields **false** if an input is **false**. The `|` operation always yields **true** if an input is **true**. Otherwise, an error-valued input causes an error-valued result. The ~ operation yields an error value when applied to an error value.

11.3.2 Numeric Operations

An integer, real, double, complex or double_complex arithmetic operation that cannot represent its result (if it is too large or too small), yields an error value. Further, a division or modulus operation having a zero divisor yields an error value. Exponentiation with both operands zero produces error.

11.3.3 Character Operations

If the input to a **character** type conversion operation is greater than 127 or less than 0, then the operation yields an error value.

11.3.4 Array Operations

An array generator having one or more error-valued parts yields an array having one or more error-valued components. If an expression in a generator size descriptor or position specifier defines an error value, the generator yields an empty error-valued array. If a size descriptor defines one or more empty dimensions, then the generator yields an empty error-valued array. If a triplet of a generator's position specifier defines a stride of zero, then the generator yields an empty error-valued array. If the intersection of a generator's position specifiers is not empty, then the duplicated elements are filled with error values. A generator failing to fill each specified position has unspecified elements filled with error values. A generator with an array part larger than its placement results in error values for the part. If all the position specifiers together are larger than the overall size descriptor, an empty error-valued array results.

Selecting an error-valued or nonexistent component of an array results in an error value, as does selecting any component of an error-valued array. Selecting a subarray of an array having one or more error valued components yields an array that might comprise one or more of those error values. Selecting a subarray of an error-valued array always results in an empty error-valued array, as does selecting a nonexistent subarray. If one or more of the expressions comprising a selection triplet define error values or the triplet's stride defines zero, then the subarray selection itself yields an empty error-valued array. The same error value semantics holds for array updates. An update that fails to fill each specified position yields error values for undefined elements, and error values result from elements specified more than once. An update that has too many replacement values for the specified area has the area filled with error values.

The size function counts both error-valued and nonerror-valued components. The size of an error-valued array is an error value. Applying size to a nonexistent dimension or error valued dimension specifier results in an error-value.

Applying the liml or limh function to an error-valued array results in an error value, as does applying either function to an error-valued dimension specifier. Applying liml or limh to a nonexistent dimension results in an error value.

Concatenating an error-valued array to any other array results in an empty error-valued array. Concatenating two arrays having one or more error values yields an array having one or more error values.

Scalar operations and predefined functions applied to error-valued arrays yield empty error-valued arrays. Otherwise, the error semantics of the particular operation or function dictates the error status of result components. The smaller parameters to dyadic and polyadic operations are extended with error values to match the larger. Taking the transpose of an array having one or more error-valued components results in an array having one or more error-valued components.

The error semantics of **for** expressions building and scattering arrays is discussed in Section 11.4.2.

11.3.5 Stream Operations

A stream generator having one or more error-valued items yields a stream having one or more error-valued constituents. If one or more expressions of an arithmetic progression define error values or the stride is zero, then the progression itself defines a single error value.

Selecting an error-valued or nonexistent component of a stream results in an error value, as does selecting any component of an error-valued stream. Selecting a substream of a stream having one or more error valued components yields a stream that might comprise one or more of those error values. Selecting a substream of an error-valued stream always results in an empty error-valued stream, as does selecting a nonexistent substream. If one or more of the expressions comprising a selection triplet define error values or the stride is zero, then the substream selection itself yields an empty error-valued stream.

Concatenating an error-valued stream to any other stream results in an empty error-valued stream. Concatenating two streams having one or more error values yields a stream having one or more error values.

Scalar operations and predefined functions applied to error-valued streams yield empty error-valued streams. Otherwise, the error semantics of the particular operation or function dictates the error status of result components. The shorter parameters to dyadic and polyadic operations are extended with error values to match the larger.

The error semantics of **for** expressions building and scattering streams is discussed in Section 11.4.2.

11.3.6 Record Operations

A record generation or replacement operation with an error valued input does not yield an error-valued record. It yields a normal record having one or more error-valued constituents.

Selecting an error-valued field results in an error value. Selecting the field of an error-valued record also results in an error value.

11.3.7 Union Operations

A union generation or replacement operation with an error-valued input does not yield an error-valued union. It yields a normal union having an error-valued constituent.

Selecting an error-valued active tag results in an error value. Selecting the tag of an error-valued union, also results in an error value. Further, attempting to select the value of an inactive tag results in an error value, as does questioning the tag activity of an error-valued union.

11.3.8 Function Operations

An expression may produce an error-valued function as its result. Attempted invocation of this function produces error values in the types that the function returns.

The result of passing an error value to a function not written in SISAL is undefined, since this error system does not extend beyond SISAL.

11.3.9 Histogram Operations

A histogram which uses an error value to specify array result size yields an empty error array of the appropriate type. A histogram built with error values to be combined yields an array with error values in the positions in which the contributing errors were reduced. A histogram which uses one or more indices which are out of bounds or which are error integers yields an array of the specified size completely filled with error values.

11.4 Errors in Compound Control Operations

In this section we defined the error semantics of SISAL's control constructs.

11.4.1 Selection Operations

When the predicate of an **if** construct yields an error value, the entire construct yields error values of the appropriate types.

Similarly, when the controlling expression of the **case** construct yields an error value, the entire construct yields error values of the appropriate types.

11.4.2 FOR Expressions

If any expression of a in-expression triplet, array scatter, or stream scatter part, or an iterative termination test yields an error value, then the **for** expression itself returns error values of the appropriate types. This is also the case if in-expressions connected with **dot** do not all have the same number of elements, or a range triplet's stride defines zero. For the "last value" part, this would be an error value of the type being returned (even for reductions). For the **array of** and **suffix** or **stream of** parts, this would be an empty error-valued array or stream, respectively.

If the expression of a masking part yields an error value, then the associated returns part yields an error value of the appropriate type.

The error semantics of the **array of** part is that of the array generator discussed in Section 11.3.4. Note that if one or more expressions of a size descriptor or **at** part of an

array of part yield error values, then the return part itself yields an empty error-valued array.

If a **for** body never executes, the default value of each `array of` part having a size descriptor is an empty error-valued array.

Chapter 12

Features Not in SISAL 2.0

This chapter discusses some features not included in SISAL 2.0, but which have value for expressibility or efficiency. They may be included when a clear design consensus emerges, or when efficient implementations are known to be possible. Researchers are welcome to comment on these ideas; exploratory prototype implementations may want to use the exemplary syntax given here.

12.1 Recursive Array Definitions

12.1.1 Motivation for Inclusion

Using an array generator to define, elementwise, an array in terms of other already defined elements of the same array is an attractive notation. For example, one might like to define an array A by giving some border values along with a formula for the other elements:

$$A_{i,j} = \begin{cases} 1 & \text{if } i = 1,\ 1 \leq j \leq n \\ 1 & \text{if } j = 1,\ 2 \leq i \leq n \\ A_{i-1,j} + A_{i-1,j-1} + A_{i,j-1} & \text{if } 2 \leq i \leq n,\ 2 \leq j \leq n \end{cases}$$

In SISAL 2.0 the first clauses are easily expressed with the array generator because the definitions are not recursive, but the last cannot be translated in a straightforward way.

In addition to expressive power, such recursive definitions can implicitly form a "wavefront" of computation that identifies the elements whose computation can occur together, in parallel. Above, all the elements in each antidiagonal are simultaneously computable.

12.1.2 Potential Difficulties

There are several problems with expressions like the last clause of the example. First, a syntactic facility to refer to the "array in progress" is needed because the array generator may not be defining a name at all; this should not be difficult to devise.

Second, the order of computing values of array elements is important. Otherwise, much machine resource can be tied up with computations which have been started and are waiting for needed data. In the example above, one should not initiate the calculation of an element before its north, northwest, and west neighbor values have been computed. Dependencies like these can define possible parallel wavefronts.

Third, some recursive array definitions can only be completed by initiating all element computations simultaneously. As a simple example, let $b_{1..n}$ contain a permutation of the integers $1..n$, and consider:

$$A_{b_i} = \begin{cases} 3 & \text{if } b_i = 1 \\ A_{i-1} & \text{if } 2 \leq b_i \leq n \end{cases}$$

See that there is no efficient evaluation order possible for this array, and these cases should be recognized and perhaps disallowed.

These difficulties have helped us to decide to forgo the inclusion of recursively defined arrays in the current definition of SISAL. However, promising research in this area is described in the next section.

12.1.3 Current Research

At least three groups are active in research towards the efficient implementation of recursive array definitions. All of them rely on restricting the forms of subscripts on already-defined array elements so a compiler can generate efficient code. We briefly review their work here.

Yale/LANL Efforts are underway to efficiently compile *non-strict* array generators.[1] However, because of the greater implementation costs of non-strictness, notation is suggested to tell a compiler that all elements are to be computed when the array is defined, that is, that the array is actually to be treated strictly. In this case, direction vectors can sometimes be obtained from subscript expressions that specify the dependencies among the recursively defined elements. This is the same technology used to vectorize expressions and loops in imperative languages, and has the same power and limitations. If direction vectors are not known, an efficient order cannot be obtained. This work is directed toward sequential evaluation of array elements.

McGill University A recent technical report[2] describes an array generator that allows recursion. In their strategy, the programmer gives a name, whose scope is the generator, to use in referencing already-defined array elements in the defining expression. The programmer is allowed to specify an order of evaluation to improve the efficiency of implementation. In the defining expression, subscripts on self-references are restricted in form to

[1] S. Anderson and P. Hudak. Compilation of Haskell Array Comprehensions for Scientific Computing. *ACM SIGPLAN Notices* 25, 6, June, 1990, pp. 137–149.

[2] Gao, G. R., R. K. Yates, J. B. Dennis and L. R. Mullin. An Efficient Monolithic Array Constructor. ACAPS Technical Memo 19, McGill University School of Computer Science, June 1990.

enable compile-time checking of the programmer-supplied evaluation order; this restriction is somewhat different than the preceding work based on vectorization. They emphasize the parallel evaluation of wavefronts.

Rensselaer Polytechnic Institute In a recent paper,[3] the optimum direction of computation in a recursively defined array is discussed. If all distance vectors in the subscript references lie on the same side of a possible wavefront, the computation is possible. In this case an optimum wavefront, perpendicular to the optimum direction of computation, can be calculated. These are promising results.

12.2 User-Defined Reductions and Nondeterminism

12.2.1 Motivations for Use

Reductions are predefined in this and earlier versions of SISAL to allow the collapsing of values produced by **for** bodies into a single result. In SISAL 2.0 function-like forms allow the direct application of a reduction to an entire array or stream as well.

The simple problem of finding a maximum array element and its index in a single **for** which has a chance to run in parallel demonstrates the desirability of providing users with the facility for defining their own reductions—no one can predict all the useful reductions beforehand. We experimented with several forms, and then settled on a method for initializing values, updating values for each reduction step, and delivering the final results. Because the reduction step resembles the value name re-assignments in an iterative **for** expression, it is also easy to write powerful reductions that solve other interesting problems. So for reasons of both expressiveness and efficiency, user-defined reductions seem to be a good idea.

12.2.2 Difficulties for Determinacy

The substantial difficulty with this design is the arbitrary computation allowed in the reduction step assignments. There is no guarantee that a programmer will write reduction steps that are associative and commutative, so the parallel use of reductions is impossible if SISAL is to continue to claim both determinacy and parallel execution as important features. Worse, there is no effective procedure to deduce, during compilation, a failure to preserve the needed properties; this would be unfortunate in a language which tries to remove much non-algorithmic burden from programmers. Hence, instead of "accidentally" introducing the potential for nondeterminism, we have removed user-defined reductions

[3] B. Sinharoy, B. McKenney and B. K. Szymanski. Scheduling EPL Programs for Parallel Processing. To appear in *Languages, Compilers and Run-time Environments for Distributed Memory Machines*, P. Mehrotra and J. Saltz, eds, Elsevier Science Publishers, 1991.

from SISAL 2.0 while the bigger questions about determinate execution are discussed on their own merits.

Debugging imperative, parallel programs is much more difficult than imperative, sequential programs. For current, determinate SISAL, one can establish the correctness of a program in a sequential setting and be assured that no errors will be introduced by moving the program to a parallel environment. The importance of this SISAL feature must not be underestimated.

However, in determinate SISAL it is not possible to express some important algorithms for parallel execution. For example, Figure 12.1 shows a parallel branch-and-bound function that needs some form of globally accessible minimum. A nondeterminate reduction as we originally designed during our deliberations is not adequate to solve this problem, since the tree-pruning occurs in different recursive threads of execution.

12.2.3 Class-Based Approach to Nondeterminism

To explore how a carefully encapsulated feature for nondeterminate computing could be added to a previously determinate language, we consider a class-based approach[4] described informally here.

A **class** is a type comprising initialization, reduction, and inspection methods or operations for a collection of values. An object of a class is first instantiated in an **instance** operation which defines a new set of these values each time the **instance** operation is referenced. The **instance** operation may have a parameter list, and returns a single value whose type is the class instantiated. This value will usually be assigned to a name which refers to this instance of the class. The initializing operation is referenced by qualifying the class name with **instance**:

```
for i in [2..size(A)];
    maxposobj := maxpos.instance(A[1], 1)
do
returns maxposobj.mp( A[i], i )
end for
```

The class also contains one or more methods called **reduction** operations, each with parameter lists and returning a single value of the same type as the instance operation (i.e., the class). Each reduction operation redefines some of the initialized (or previously redefined) names in an atomic fashion. A reduction operation is referenced by qualifying the object *instance* name with the reduction name.

Finally, each class has a **returns** operation which names the values to be supplied by a reference to the operation (not all initialized or updated names need to be returned). One does not explicitly use this operation when, as above, the reductions are part of the

[4] A good taxonomy of terms related to this is found in "Concepts and Paradigms of Object-Oriented Programming" by Peter Wegner, in *OOPS Messenger* 1, 1 (August 1990).

```
function bb( W, P: array of integer, i, CW, CP, M, n: integer
             returns integer);

% i < n, W: weights, P: profits, CW (<= M) : cum weight, CP: cum profit

   let U, L := for j in [i+1..n]; CL := CP do
                  CW := CW + W[j];
                  CP, CL :=
                     if CW <= M then CP + P[j], CL + P[j]
                     else CP + (1 - (CW - M) / W[j]) * P[j], CL
                     end if
               until CW > M
               returns CP, CL
               end for;

% THIS NEXT LINE IS NOT SISAL!
       GL := max( GL, L )

   in
       if U < GL then -maxint
       else
          if i = n-1 then L
          elseif CW + W[i] > M then bb(W, P, i+1, CW, CP, M, n)
          else max( bb(W, P, i+1, CW, CP, M, n),
                    bb(W, P, i+1, CW+W[i], CP+P[i], M, n) )
          end if
       end if
   end let
end function
```

Figure 12.1: Parallel branch-and-bound needs global state

```
type maxpos = class

                    instance( fval, fpos: integer returns maxpos);
                       val := fval;
                       pos := fpos;

                    reduction mp( nv, np: integer returns maxpos);
                       val, pos := if nv > val then nv, np
                                   else val, pos
                                   endif;

                    returns val, pos;

              end class
```

Figure 12.2: Class Type Definition for Maximum Value and Position.

gathering process in a **returns** part of a **for** expression. When used, **returns** qualifies an object name and "unwraps" the values returned by the object.

The infix operator `&` may be used to combine the results of reduction operations on the *same* instance of a class; the result is the object instance itself, with both reductions applied.

The class definition for the above example is in Figure 12.2.

The branch-and-bound example is expressed in Figure 12.3 using a class type `globmax` which includes the reduction `gmax`, the **returns** operator for inspection of intermediate values as well as for obtaining the final result, and the `&` operation.

Since objects are, frankly, collections of values with persistent state, it should come as no surprise that their semantics are different from other SISAL values. Once instantiated, an object may have reductions applied to it, each of which results in the *same* object that the reductions updated. An object may be passed to or returned from functions, but the effect is passing and return by reference.

Finally, a SISAL program is still guaranteed to be determinate if the classes described here are not used. In some cases the user-defined objects may preserve determinacy, but establishing that this is so is in the province of the programmer, not SISAL software.

```
type globmax = class

                  instance( v: integer returns globmax );
                     currmax := v;

                  reduction gmax( v: integer returns globmax );
                     currmax := if currmax < v then v
                                else currmax
                                end if;

                  returns currmax;

               end class

. . .

function bb( W, P: array of integer, i, CW, CP, M, n: integer,
             bbmax: globmax returns globmax );

   let U, L := . . . ;
       GL   := bbmax.gmax( L )
   in
       if U < GL.returns then GL
       else if i = n-1 then GL
            elseif CW + W[i] > M then
               bb( W, P, i+1, CW, CP, M, n, GL )
            else let m1 := bb( W, P, i+1, CW, CP, M, n, GL );
                     m2 := bb( W, P, i+1, CW+W[i], CP+P[i], M, n, GL )
                 in  m1 & m2
                 end let
            end if
       end if
   end let

end function

. . .

let
    bbmax := globmax.instance( -maxint )
in
    (bb(W, P, 1, CW, CP, M, n, bbmax)).returns
end let
```

Figure 12.3: Parallel Branch-and-Bound Solution Using Objects.

12.3 Interactive Input/Output

12.3.1 Motivation—Interactive Programs

Input/output in functional languages can present difficulties in preserving referential transparency. A stream-based approach for some values which interface with an operating system environment is a successful approach for SISAL and other similar languages.

Although the detailed operation of input/output is not defined in this manual, the most common implementation works according to the following:

Input: Non-stream parameters of the function where execution begins are assumed to be values available in external Fibre format from standard input. Values are read and assigned to the formal parameters. Parameters which are streams are associated with files in the environment of execution. Operations to consume the stream parameters result in input file reading.

Output: Non-stream result values of the function where execution began are written, in Fibre format, on standard output when the function evaluation is complete. Result values which are streams are associated with files in the execution environment. Stream production results in output file writing.

This has been effective for many programs that do not require interaction to execute.

12.3.2 Current Difficulties

To write an interactive SISAL program, one could use a stream of characters for an input parameter and a result respectively, and associate these streams with standard input and output rather than named files. However, this has at least two problems:

- The nonstrict nature of streams means that the interactive dialogue one wants to program is not guaranteed to have the order of intermixing of prompt and response preserved. In many systems one can distinguish whether a file is standard input or output as opposed to an actual file, and make the input and output operations strict. So this problem can be solved.

- To make responses readable by people, internal non-character values must be translated to external character string form. Further, all sensible languages provide convenient formatting capabilities on the output side. While this can be done with functions from a module of such utilities, the familiarity and convenience of a "format string" is economical.

12.3.3 Predefined Functions

We propose the experimental addition of two "functions" to the SISAL language to explore their effectiveness in adding a useful interactive input/output capability. They differ from user-defined SISAL functions in having a variable number of parameters or result values. They are modeled on the ANSI C functions `fprintf` and `fscanf`.

```
type iostream = stream of character;
type format   = array of character;
function fprintf( s: iostream, f: format, ... returns iostream );
function fscanf(  s: iostream, f: format returns iostream, ... );
```

The model for these functions is the similar functionality found in ANSI C. The function `fprintf` accepts a stream of characters that has already had zero or more of output characters produced upon it, an array of characters that resembles a formatting string in C, and zero or more values. The values are converted to character strings under the direction of the format string and are appended to form a new character stream that is the value of `fprintf`.

The function `fscanf` accepts a character stream value that is one of the parameters of the "main" SISAL function, and a *constant* format string. The result is the character string like the input with some characters consumed, and zero or more values whose types are known at compile time from the format string.

Since input values are not delimited as in Fibre format, only scalars may be obtained via `fscanf`. Similarly, since C format strings are incapable of describing the output of arrays, `fprintf` accepts only scalar values for output. In either case, iteration can obtain arrays and streams from interactive input or emit array and stream values.

Appendix A

Syntax Summary

compilation-unit ::= interface | module

interface ::= **interface** interface-id [{ **for** | **in** } language-id]
[import-decl | type-def | int-function-decl] ⋯
end interface

module ::= { **program** | **module** } module-id
[import-decl | type-def | function-def] ⋯
end { **program** | **module** }

import-decl ::= **from** interface-id [: import-id [, import-id] ⋯] [;]

import-id ::= type-id | function-id

type-def ::= **type** type-id [= type-spec] [;]
| **type state** [;]

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

type-spec ::= basic-type
| compound-type
| type-set
| [interface-id .] type-id
| **type** (expression)

type-set ::= type-spec [{ + | - } type-spec] ⋯

basic-type ::= **boolean**
| **character**
| **integer**

	|	**real**
	|	**double**
	|	**complex**
	|	**double_complex**
	|	**null**
	|	**foreign** language-id **function**
compound-type	::=	array-spec
	|	stream-spec
	|	record-spec
	|	union-spec
	|	function-spec
array-spec	::=	**array** [dim-spec] **of** type-spec
stream-spec	::=	**stream of** type-spec
record-spec	::=	**record** [field-spec [; field-spec] · · · [;]]
union-spec	::=	**union** [tag-spec [; tag-spec] · · · [;]]
function-spec	::=	**function** ([type-spec-list] **returns** type-spec-list)
tag-spec	::=	tag-id [, tag-id] · · · : type-spec
field-spec	::=	field-id [, field-id] · · · : type-spec
int-function-decl	::=	**function** function-id ([int-param-decls] [**returns** type-spec-list]) [;]
int-param-decls	::=	int-param-decl [, int-param-decl] · · ·
int-param-decl	::=	[**in** | **out** | **inout**] type-spec
function-header	::=	function-id ([param-decls] **returns** type-spec-list)
param-decls	::=	param-decl [, param-decl] · · · [,]
param-decl	::=	value-id [, value-id] · · · : type-spec
function-def	::=	**forward function** function-id ([type-spec-list] **returns** type-spec-list) [;]

```
                 |    function function-header
                      expression-list
                      end function

unary-op         ::=  + | - | ~

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

expression-list  ::=  expression [, expression ] · · ·

expression       ::=  expression binary-op expression
                 |    primary

primary          ::=  ( expression )
                 |    unary-op expression
                 |    [ interface-id . ] function-id
                 |    reduction-id
                 |    value-id
                 |    constant
                 |    invocation
                 |    error-test
                 |    union-ref
                 |    union-test
                 |    union-gen
                 |    record-ref
                 |    record-gen
                 |    stream-ref
                 |    stream-gen
                 |    array-ref
                 |    array-gen
                 |    array-update
                 |    complex-ref
                 |    complex-gen
                 |    conditional-exp
                 |    let-exp
                 |    case-exp
                 |    for-exp
                 |    type-spec

invocation       ::=  primary ( [ [ expression ] [, [ expression ] ] · · · ] )

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

```
complex-ref       ::=  primary . { imag | real }

complex-gen       ::=  ( expression , expression )

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

union-ref         ::=  primary . tag-id

union-gen         ::=  union type-spec [ tag-id : expression [;] ]
                   |   primary [ tag-def [;] ]

tag-def           ::=  tag-id [. tag-id ]··· : expression

record-ref        ::=  primary . field-id

record-gen        ::=  record type-spec [ field-def [; field-def ]··· [;] ]
                   |   primary [ field-def [; field-def ]··· [;] ]

field-def         ::=  field-id [. field-id ]··· : expression

dim-spec          ::=  [ .. [, .. ]··· ]

size-descriptor   ::=  [ value-id in ] triplet [, [ value-id in ] triplet ]···

triplet           ::=  [ expression ] .. [ expression ] [.. expression ]

diag-spec         ::=  value-id dot value-id [ dot value-id ]···

selector          ::=  selector-part [, selector-part ]···
                             [ { diag-spec [, diag-spec ]··· } ]

selector-part     ::=  expression | [ value-id in ] triplet

array-ref         ::=  primary [ selector ]

array-update      ::=  primary [ update-part [; update-part ]··· [;] ]

update-part       ::=  selector : expression-list

array-gen         ::=  array type-spec
                       [ [ [ size-descriptor : ] array-part [; array-part ]··· [;] ] ]

array-part        ::=  [ placement ] expression-list
```

placement	::=	[{ selector | **otherwise** }]
stream-ref	::=	primary [{ triplet | expression }]
stream-gen	::=	**stream** type-spec [[stream-part [, stream-part] · · ·]]
stream-part	::=	expression
	|	triplet
conditional-exp	::=	**if** expression **then** expression-list
		[**elseif** expression **then** expression-list] · · ·
		else expression-list
		end if
let-exp	::=	**let** decldef-part **in** expression-list **end let**
case-exp	::=	**case** expression **of**
		case-part [; case-part] · · ·
		[; **otherwise** : expression-list] [;]
		end case
case-part	::=	pattern [, pattern] · · · : expression-list
pattern	::=	**false**
	|	**true**
	|	integer-constant
	|	character-constant
	|	character-string-constant
	|	tag-id
	|	type-spec
for-exp	::=	[for-top]
		for-body
		for-bottom
for-top	::=	**for** in-exp-list [; [decldef-part]] [for-test]
	|	**for** decldef-part [for-test]
	|	for-test
for-body	::=	**do** [decldef-part] [for-test]
for-bottom	::=	**returns** return-exp [, return-exp] · · ·
		end { **for** | **while** | **until** | **do** }

for-test	::=	{ **while** | **until** } expression
return-exp	::=	**stream of** expression [filter]
	|	expression [filter]
	|	**suffix** (value-id)
	|	**array** [[[size-descriptor]]] **of** expression
		[filter | **at** [at-exp [, at-exp] · · ·]]
filter	::=	**when** expression
	|	**unless** expression
at-exp	::=	expression | ..
in-exp-list	::=	in-exp [**dot** in-exp] · · ·
	|	in-exp [**cross** in-exp] · · ·
in-exp	::=	value-id **in** expression [**at** [index-id-list]]
	|	value-id **in** [triplet]
index-id-list	::=	{ .. | value-id } [, { .. | value-id }] · · ·
decldef-part	::=	decldef [; decldef] · · · [;]
decldef	::=	decl [, decl] · · · := expression-list
decl	::=	decl-id [, decl-id] · · · [: type-spec]
decl-id	::=	value-id
	|	function-header
constant	::=	**false**
	|	**true**
	|	**nil**
	|	integer-constant
	|	real-constant
	|	double-constant
	|	complex-constant
	|	double_complex-constant
	|	character-constant
	|	character-string-constant
	|	**error** [type-spec]
interface-id	::=	name

module-id	::=	name
language-id	::=	name
function-id	::=	name
reduction-id	::=	name
type-id	::=	name
tag-id	::=	name
field-id	::=	name
value-id	::=	name

Appendix B

Implementation Notes

B.1 Limitations

An implementation may choose to restrict the base type of streams for input and output purposes to those whose sizes are known at compile time. This means that the base type of a stream returned from any function defined in the program module could be prohibited from being an array or stream, or a record or union type containing arrays or streams as part of their definition.

An implementation may choose to recognize the ASCII form feed character (octal 14) as white space in source text.

B.2 Pragmas

If the first character of a comment is a dollar sign, the comment is a compiler directive, or pragma. A pragma is an annotation to a program that gives the compiler information about how the programmer intends the associated language unit to be processed. The absence or presence of pragmas can never influence the results from a program, but may have effects on its efficiency. However, a pragma that influences the order of application of a predefined reduction on floating point values may cause differences in the computed result. The syntax for a pragma is

pragma ::= `%$name`[`=text`]

where `name` is a sequence of letters, digits, and underscores, of which the first must be a letter, and `text` is a sequence of characters between '!' and '~' in the ASCII sequence (inclusive). White space (a tab, space, or newline character) ends a pragma. The `text`, which is optional, provides further information about the pragma (implemenation dependent).

The compiler associates each pragma with the first reserved word, predefined function, constant, name, or arithmetic symbol following it in the source text. For example, in

```
%$slice
for i in [1..10000] do returns sum(i) end for
```

the compiler associates pragma `slice` with the **for** reserved word; hence the **for** expression itself. Note that it is possible to provide more than one pragma on the same source line:

```
%$slice %$chunksize=10
for i in [1..10000] do returns sum(i) end for
```

Here the compiler will associates pragmas `slice` and `chunksize` with the **for** expression.

Pragma semantics is implementation dependent, although we encourage standardization to facilitate software sharing.

B.3 Errors

An implementation may choose to stop execution when an error is encountered instead of producing an error value. However, if error values are to be generated and propagated, they must behave as specified in Chapter 11.

Appendix C

Sample Programs

In this appendix, to illustrate the use of SISAL, we present several sample programs: Quicksort, Sieve of Eratosthenes, Gaussian Elimination, Matrix Multiplication, and the Game of Life.

C.1 Quicksort

```
program QuickSortExample

% This program sorts an input array of integers using a recursive
% implementation of quicksort.

  type Info = array of integer;

  function QuickSort( Data:Info returns Info );

     if( size( Data ) < 2 then Data
     else let Pivot := Data[ liml(Data) ]
              Low, Mid, High := for E in Data do
                                returns array of E when E < Pivot,
                                        array of E when E = Pivot,
                                        array of E when E > Pivot
                                end for
          in
              QuickSort( Low ) || Mid || QuickSort( High )
    end if
  end function

end program
```

C.2 Sieve of Eratosthenes

```
% This program returns the primes between 2 and Limit using the Sieve of
% Eratosthenes.

interface MathF77 in FORTRAN
  function dsqrt( a:double returns double );
end interface

Program PrimesExample

  from MathF77: dsqrt;

  type StrmInt = stream of integer;

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

  function Sieve( Limit:integer returns StrmInt )
    stream integer [2] ||
    for
      S :=  stream integer [ 3..Limit..2 ];
      Maxt := integer( dsqrt( double( Limit ) ) );
    until empty( S ) do
      T := S[1];
      S := if T <= Maxt then Filter( S[2..], T )
           else S[2..] end if;
    returns stream of T
    end for
  end function

end program
```

C.3 Gaussian Elimination

```
program GaussExample

% This program solves A*x=B for x.

  type OneI   = array [..] of integer;
  type OneD   = array [..] of double;
  type TwoD   = array [..] of OneD;


  function Reduce( pivot:integer, A:TwoD, B:OneD
                   returns TwoD, OneD )
    let
      mults := A[..,pivot] / A[pivot,pivot];
    in
      for row in A at [i] do
        nrow,
        nB := if i = pivot then
                row  / A[pivot,pivot],
                B[i] / A[pivot,pivot]
              else
                row  - mults[i] * A[pivot],
                B[i] - mults[i] * B[pivot]
              end if
      returns array of nrow,
              array of nb
      end for
    end let
  end function


  function Main( n:integer, A:TwoD, B:OneD returns OneD )
    for i in [1..n] do
      A, B := Reduce( i, A, B );
    returns B
    end for
  end function

end program
```

C.4 Matrix Multiplication

```
% This program illustrates the definition and use of a matrix package
% comprising a matrix multiplication operation.

interface MatrixRoutines
  type TwoDim = array [..,..] of numeric;
  function MatMult( A, B: TwoDim returns TwoDim );
end interface

module MatrixRoutines

  function Matmult( A, B: TwoDim returns TwoDim )
     if size(A,2) ~= size(B,1) then error[TwoDim]
     else array numeric [i in 1..size(A,1), j in 1..size(B,2):
                  [i,j] sum(A[i,..] * B[..,j])]
     endif
  end function

end module

program MatrixMultiplyExample

  from MatrixRoutines: MatMult;

  type TDR = array[..,..] of real;

  function MatMult( A, B: TDR returns TDR )
    MatrixRoutines.MatMult( A, B )
  end function

end program
```

C.5 Game of Life

```
program GameOfLifeExample

% John Conway's Game of Life.  Values of the Grid are 0's or 1's.  A
% cell has 8 neighbors.  Each iteration updates cells as follows:
%  --If a cell has a 1 and greater than five of its neighbors have
%   1's, then it should be updated to a 0.
%  -- If a cell has a 0 and has more than three or fewer then three
%   neighbors have 1's, then it should be updated to a 1.
%  -- otherwise the value of a cell remains unchanged.
% The simulation iterates n times. The border of the grid never
% changes. The lower bound of each dimension is assumed to be 1.

  type  Grid = array[..,..] of integer;

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

  function DoWork( G:Grid, r,c:integer returns Grid )
    let
      M := for k in [2..r-1] cross j in [2..c-1] do
           returns array[2..r-1,2..c-1] of Compute(G,k,j)
           end for;
    in
      G[2..r-1,2..c-1: [..,..] M ]
    end let
  end function

  function Life( n:integer, G:Grid returns Grid )
    let
      r  := size( G, 1 );
      c  := size( G, 2 );
    in
      for i in [1..n] do
```

```
        G := DoWork( G, r,c );
      returns G
      end for
    end let
  end function

end program
```

Appendix D

FIBRE Format for External Values

A SISAL program reads data for the values of formal parameters at the function where execution begins. The result values defined there are written when execution of this function is complete. The FIBRE format is used for textual (ASCII) input and output from SISAL programs.

FIBRE syntax is similar to SISAL syntax, but not identical. For example, FIBRE arrays use white space to separate values, FIBRE records do not include field identifiers, and FIBRE unions use ordinal values for tag identifiers. These are in contrast to commas for separating values in array constructions, identifiers for designating fields in record definitions, and identifiers for designating tags in union definitions. Because FIBRE avoids record and union field identifiers, a symbol table is not required.

In this appendix, we use the BNF notation defined in Chapter 2. The complete BNF for FIBRE follows:

fibre-unit ::= [simple-value | compound-value] · · ·

simple-value ::= [sign] integer | [sign] real | complex | character |
boolean | null

compound-value ::= array | stream | record | union

sign ::= `+` | `-`

integer ::= digit[digit] · · ·

power ::= real-power | double-power
real-power ::= `e` | `E`
double-power ::= `d` | `D`

real ::= integer[.]power[sign]integer

	|	integer . [integer][power[sign]integer]
character	::=	’char’
char	::=	printable-character | non-printable-character
boolean	::=	**true** | **false**
null	::=	**nil**
complex	::=	([sign] real , [sign] real)
array	::=	[[size-descriptor : value [value] ···]]
	|	string
size-descriptor	::=	integer .. integer [, integer .. integer] ···
string	::=	"[char] ···"
stream	::=	{ [value] ··· }
record	::=	< value [value] ··· >
union	::=	(integer : value)

The remainder of this appendix examines the individual FIBRE constructs in more detail.

D.1 Line Format and Comments

FIBRE is free-format. Multiple occurrences of “white space” (blanks, tabs, and newlines, and comments) are allowed between values and delimiters. Comments in FIBRE are syntactically identical to those in SISAL. That is, a comment begins with a percent sign and ends with a newline character. Comments, strings and simple values cannot cross line boundaries.

D.2 Simple Values

Simple values have the same syntax as SISAL literals (Chapter 2). The simple values have type integer, real, double, complex, double_complex, character, boolean, or null.

simple-value	::=	[sign] integer | [sign] real | complex | character |

		boolean | null
sign	::=	`+` | `-`

D.2.1 Integer

integer	::=	digit[digit] $\cdots$

Integer values describe data of type integer. They are non-empty sequence of decimal numerals; for example, 0 and 2010.

D.2.2 Real

power	::=	real-power | double-power
real-power	::=	`e` | `E`
double-power	::=	`d` | `D`
real	::=	integer[`.`]power[sign]integer
	|	integer`.`[integer][power[sign]integer]

Floating point values can be of either single or double precision. They describe data of type real or double respectively. The power symbol in the exponent field makes the distinction. A double precision floating point value must have an exponent field with either a 'D' or 'd' symbol. All other floating point values (those without exponents or with exponents having either an 'e' or 'E' symbol) are of single precision. Both fractions and exponents are in base ten. Consider the following examples:

34d0	% double precision
34.e10	% single precision
0.14159	% single precision
301.4E-64	% single precision
301.4D-64	% double precision

D.2.3 Complex

complex	::=	`(` [sign] real `,` [sign] real `)`

Complex values describe data of type complex or double_complex. Each is represented by two comma-separated real numbers enclosed in parenthesis. The first real number is the real part and the second is the imaginary part. For example,

```
( 2.34, -8.9 )
```

is a complex number with `2.34` as its real part and `-8.9` as its complex part.

D.2.4 Character

character	::=	'char'
char	::=	printable-character | non-printable-character

Character values describe data of type character. They are single printable characters or non-printable characters enclosed in single quotes. A printable character can be any ASCII character between blank (' ') and tilde ('~'), inclusive. Using backslash notation, a character value may describe any non-printing ASCII character (referred to as a control character). A backslash 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	for use in string constants
\'	apostrophe	for use in a character constants

A backslash followed by any other character besides the first of three consecutive octal digits, is simply the character; that is, '\s' is equivalent to the character value 's'. A backslash followed by three octal digits represents the ASCII character having the octal value. For example, the character value '\101' is the character value 'A'. This format also allows the representation of control characters. For example, '\011' represents horizontal tab.

Additional examples follow:

'a'	% little-a
'\\'	% backslash
'\n'	% newline
'\001'	% control-a

D.2.5 Boolean

boolean ::= **true** | **false**

These data values have type boolean. They are represented by either the symbol **true** or **false**. Note that FIBRE does not distinguish between upper and lower case, as does SISAL, so **TrUe** and **true** are equivalent.

D.3 Compound Values

In FIBRE, the syntax of compound values differs from SISAL. The compound values are:

compound-value ::= array | stream | record | union

FIBRE does not allow the description of functions or values containing functions (see Chapter 9).

D.3.1 Arrays

array	::=	`[` [size-descriptor : value [value] ···] `]`
	|	string
size-descriptor	::=	integer .. integer [, integer .. integer] ···
string	::=	"[char] ···"

Array values describe data of type array. Unless they take the form of strings or empty one-dimensional arrays, array values must provide both lower and upper bound pairs for each dimension, and the upper bound must be at least as large as the lower bound. The assumed lower bound of a string or empty array is 1.

The following FIBRE value describes an array of 11 characters:

```
[ 1..11: 'h' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' ]
```

One can also represented it as a string:

```
"hello world"
```

Strings describe data of type array of character. They always have upper bounds equal to the number of printable and non-printable characters they contain (refer to Chapter 2).

The following array value describes a matrix of double precision floating point numbers:

```
[ 1..4, 2..4:
    1D0 2D0 3D0
    4D0 5D0 6D0 7D0
    8D0 9D0
    1D1 1D2 1D3
    ]
```

This matrix has four rows of three elements. The constituents are provided in increasing subscript order, with the subscript of the first dimension increasing the slowest. Hence in this example, 7D0 is the first element of the third row, which contains 7D0, 8D0, and 9D0.

The following value describes a three dimensional array of one dimensional arrays (describing date of type "array [..,..,..] of array of boolean"):

```
[ 1..1, 1..2, 1..1:
     [ 2..5: true false true true ]
     [ 1..2: true false ]
     ]
```

Note that the second component, an array containing two booleans, is at position [1,2,1] within the value.

D.3.2 Streams

stream ::= { [value] · · · }

These are data of type stream. They are sequences of zero or more values delimited by { and }. The lower bound of a stream is always one, and never specified. A stream value of ten prime numbers is

```
{ 2 3 5 7 11 13 17 19 23 29 }
```

D.3.3 Records

record ::= < value [value] · · · >

These data values have type record. They are ordered list of values delimited by < and >. The order of the fields in a record must match the order of fields in the associated type. For example, assuming the type declaration

```
type Name = record[
               First,Last:array of character;
               Age:integer
               ];
```

the FIBRE record for John Doe, age 12, is

```
<
  "John"
  "Doe"
  12
>
```

D.3.4 Unions

union	::=	(integer : value)
null	::=	**nil**

These data have type union. They are pairs of values delimited by `(` and `)`. The first value identifies the tag (its ordinal value) and the second describes the tag's associated data. The ordinals begin at zero, and are assigned in the order in which the tags appear in the associated type. Null, a type normally used in union field declarations, has the value **nil**. Again **nil**, **nIL**, **Nil**, etc. are equivalent.

To illustrate union values we will build a FIBRE object of type

```
type List = union[
               Empty:null;
               Occupied : record[ Data:integer Next:List ];
               ];
```

Now the list (-6, 0, 15) is described as

```
( 1: < -6 ( 1: < 0 ( 1: 15 ( 0: nil ) ) > ) > )
```

This list comprises four nested union values. The tag `Empty` has ordinal zero, and the tag `Occupied` has ordinal one (see the order of tag declarations within `List` above).

D.4 Error Values

FIBRE uses the symbol **error** to describe simple values in error (regardless of type).

Compound values preceded by the symbol **error** are in error. For example, an error array of integers is described by:

```
error[ 1..8: 1 2 3 4 5 6 7 8 ]
```

The other compound values have similar descriptions. Note that white space is allowed between `error` and the left delimiter of the compound value.

The compound value might contain values that are in error. For example, the third field in the following record description is in error:

```
< 0 (4: 3.445d-10) error 30 >
```

If a FIBRE value read by a program does not provide enough values, the missing values are automatically assumed to be in error. For example,

```
[ 1..4, 2..3: 1.0 2.0 3.0 ]
```

is missing five values. The FIBRE processing system assumes that this value really defines the following array value:

```
[ 1..4, 2..3:
    1.0   2.0
    3.0   error
    error error
    error error
    ]
```

Also, if

```
< 0.0D0 '\n' >
```

describes data of type

```
record[ first:double; second:character; third:array of integer ]
```

then it really defines

```
< 0.0D0 '\n' error[] >
```

That is, the shape of the missing array is unknown. If a compound value read by a program provides more constituents than expected, the FIBRE processing system must provide at least a warning to the user.

Index