Chapel Language Specification 0.796

October 21, 2010

Cray Inc
901 Fifth Avenue, Suite 1000
Seattle, WA 98164

Contents

1 Scope

Chapel is a new parallel programming language that is under development at Cray Inc. in the context of the DARPA High Productivity Computing Systems initiative.

This document is ultimately intended to be the definitive specification of the Chapel language. The current draft is a work-in-progress and therefore incomplete.

2 Notation

Special notations are used in this specification to denote Chapel code and to denote Chapel syntax.

Chapel code is represented with a fixed-width font where keywords are bold and comments are italicized.

> *Example*.
>
> ```
> for i in D do   // iterate over domain D
>   writeln(i);   // output indices in D
> ```

Chapel syntax is represented with standard syntax notation in which productions define the syntax of the language. A production is defined in terms of non-terminal (*italicized*) and terminal (non-italicized) symbols. The complete syntax defines all of the non-terminal symbols in terms of one another and terminal symbols.

A definition of a non-terminal symbol is a multi-line construct. The first line shows the name of the non-terminal that is being defined followed by a colon. The next lines before an empty line define the alternative productions to define the non-terminal.

> *Example*. The production
>
> *bool-literal*:
>   **true**
>   **false**
>
> defines *bool-literal* to be either the symbol **true** or **false**.

In the event that a single line of a definition needs to break across multiple lines of text, more indentation is used to indicate that it is a continuation of the same alternative production.

As a short-hand for cases where there are many alternatives that define one symbol, the first line of the definition of the non-terminal may be followed by "one of" to indicate that the single line in the production defines alternatives for each symbol.

> *Example*. The production
>
> *unary-operator*: *one of*
>   + − ˜ !
>
> is equivalent to
>
> *unary-operator*:
>   +
>   −
>   ˜
>   !

As a short-hand to indicate an optional symbol in the definition of a production, the subscript "opt" is suffixed to the symbol.

Example. The production

formal:
  formal−tag identifier formal−type$_{opt}$ *default−expression*$_{opt}$

is equivalent to

formal:
  formal−tag identifier formal−type default−expression
  formal−tag identifier formal−type
  formal−tag identifier default−expression
  formal−tag identifier

3 Organization

This specification is organized as follows:

- Chapter 1, Scope, describes the scope of this specification.
- Chapter 2, Notation, introduces the notation that is used throughout this specification.
- Chapter 3, Organization, describes the contents of each of the chapters within this specification.
- Chapter 4, Acknowledgments, offers a note of thanks to people and projects.
- Chapter 5, Language Overview, describes Chapel at a high level.
- Chapter 6, Lexical Structure, describes the lexical components of Chapel.
- Chapter 7, Types, describes the types in Chapel and defines the primitive and enumerated types.
- Chapter 8, Variables, describes variables and constants in Chapel.
- Chapter 9, Conversions, describes the legal implicit and explicit conversions allowed between values of different types. Chapel does not allow for user-defined conversions.
- Chapter 10, Expressions, describes the non-parallel expressions in Chapel.
- Chapter 11, Statements, describes the non-parallel statements in Chapel.
- Chapter 12, Modules, describes modules in Chapel., Chapel modules allow for name space management.
- Chapter 13, Functions, describes functions and function resolution in Chapel.
- Chapter 14, Tuples, describes tuples in Chapel.
- Chapter 15, Classes, describes reference classes in Chapel.
- Chapter 16, Records, describes records or value classes in Chapel.
- Chapter 17, Unions, describes unions in Chapel.
- Chapter 18, Ranges, describes ranges in Chapel.
- Chapter 19, Domains, describes domains in Chapel. Chapel domains are first-class index sets that support the description of iteration spaces, array sizes and shapes, and sets of indices.
- Chapter 20, Arrays, describes arrays in Chapel. Chapel arrays are more general than in most languages including support for multidimensional, sparse, associative, and unstructured arrays.
- Chapter 21, Iterators, describes iterator functions.
- Chapter 22, Generics, describes Chapel's support for generic functions and types.
- Chapter 23, Input and Output, describes support for input and output in Chapel, including file input and output..
- Chapter 24, Task Parallelism and Synchronization, describes task-parallel expressions and statements in Chapel as well as synchronization constructs, atomic sections, and memory consistency.

- Chapter 25, Data Parallelism, describes data-parallel expressions and statements in Chapel including reductions and scans, whole array assignment, and promotion.
- Chapter 26, Locales, describes constructs for managing locality and executing Chapel programs on distributed-memory systems.
- Chapter 27, Domain Maps, describes Chapel's *domain map* construct for defining the layout of domains and arrays within a single locale and/or the distribution of domains and arrays across multiple locales.
- Chapter 28, User-Defined Reductions and Scans, describes how Chapel programmers can define their own reduction and scan operators.
- Chapter 29, User-Defined Domain Maps, describes how Chapel programmers can define their own domain maps to implement domains and arrays.
- Chapter 30, Standard Modules, describes the standard modules that are provided with the Chapel language.
- Chapter 31, Standard Distributions, describes the standard distributions (multi-locale domain maps) that are provided with the Chapel language.
- Chapter 32, Standard Layouts, describes the standard layouts (single locale domain maps) that are provided with the Chapel language.
- Appendix A, Collected Lexical and Syntax Productions, contains the syntax productions listed throughout this specification in both alphabetical and depth-first order.

4 Acknowledgments

The following people have contributed to the design of the Chapel language: Robert Bocchino, David Callahan, Bradford Chamberlain, Sung-Eun Choi, Steven Deitz, Roxana Diaconescu, James Dinan, Samuel Figueroa, Shannon Hoffswell, Mary Beth Hribar, David Iten, Mark James, Mackale Joyner, Jacob Nelson, John Plevyak, Lee Prokowich, Albert Sidelnik, Andy Stone, Greg Titus, Wayne Wong, and Hans Zima.

Chapel is a derivative of a number of parallel and distributed languages and takes ideas directly from them, especially the MTA extensions of C, HPF, and ZPL.

Chapel also takes many serial programming ideas from many other programming languages, especially C#, C++, Java, Fortran, and Ada.

The preparation of this specification was made easier and the final result greatly improved because of the good work that went in to the creation of other language standards and specifications, in particular the specifications of C# and C.

5 Language Overview

Chapel is a new programming language under development at Cray Inc. as part of the DARPA High Productivity Computing Systems (HPCS) program to improve the productivity of parallel programmers.

This section provides a brief overview of the Chapel language by discussing first the guiding principles behind the design of the language and second how to get started with Chapel.

5.1 Guiding Principles

The following four principles guided the design of Chapel:

1. General parallel programming
2. Locality-aware programming
3. Object-oriented programming
4. Generic programming

The first two principles were motivated by a desire to support general, performance-oriented parallel programming through high-level abstractions. The second two principles were motivated by a desire to narrow the gulf between high-performance parallel programming languages and mainstream programming and scripting languages.

5.1.1 General Parallel Programming

First and foremost, Chapel is designed to support general parallel programming through the use of high-level language abstractions. Chapel supports a *global-view programming model* that raises the level of abstraction of expressing both data and control flow when compared to parallel programming models currently used in production. A global-view programming model is best defined in terms of *global-view data structures* and a *global view of control*.

Global-view data structures are arrays and other data aggregates whose sizes and indices are expressed globally even though their implementations may distribute them across the *locales* of a parallel system. A locale is an abstraction of a unit of uniform memory access on a target architecture. That is, within a locale, all threads exhibit similar access times to any specific memory address. For example, a locale in a commodity cluster could be defined to be a single core of a processor, a multicore processor or an SMP node of multiple processors.

Such a global view of data contrasts with most parallel languages which tend to require users to partition distributed data aggregates into per-processor chunks either manually or using language abstractions. As a simple example, consider creating a 0-based vector with n elements distributed between p locales. A language like Chapel that supports global-view data structures allows the user to declare the array to contain n elements and to refer to the array using the indices $0 \ldots n-1$. In contrast, most traditional approaches require the user to declare the array as p chunks of n/p elements each and to specify and manage interprocessor communication and synchronization explicitly (and the details can be messy if p does not divide

n evenly). Moreover, the chunks are typically accessed using local indices on each processor (*e.g.*, $0..n/p$), requiring the user to explicitly translate between logical indices and those used by the implementation.

A *global view of control* means that a user's program commences execution with a single logical thread of control and then introduces additional parallelism through the use of certain language concepts. All parallelism in Chapel is implemented via multithreading, though these threads are created via high-level language concepts and managed by the compiler and runtime, rather than through explicit fork/join-style programming. An impact of this approach is that Chapel can express parallelism that is more general than the Single Program, Multiple Data (SPMD) model that today's most common parallel programming approaches use as the basis for their programming and execution models. Chapel's general support for parallelism does not preclude users from coding in an SPMD style if they wish.

Supporting general parallel programming also means targeting a broad range of parallel architectures. Chapel is designed to target a wide spectrum of HPC hardware including clusters of commodity processors and SMPs; vector, multithreading, and multicore processors; custom vendor architectures; distributed-memory, shared-memory, and shared address space architectures; and networks of any topology. Our portability goal is to have any legal Chapel program run correctly on all of these architectures, and for Chapel programs that express parallelism in an architecturally-neutral way to perform reasonably on all of them. Naturally, Chapel programmers can tune their codes to more closely match a particular machine's characteristics, though doing so may cause the program to be a poorer match for other architectures.

5.1.2 Locality-Aware Programming

A second principle in Chapel is to allow the user to optionally and incrementally specify where data and computation should be placed on the physical machine. Such control over program locality is essential to achieve scalable performance on large machine sizes. Such control contrasts with shared-memory programming models which present the user with a flat memory model. It also contrasts with SPMD-based programming models in which such details are explicitly specified by the programmer on a process-by-process basis via the multiple cooperating program instances.

5.1.3 Object-Oriented Programming

A third principle in Chapel is support for object-oriented programming. Object-oriented programming has been instrumental in raising productivity in the mainstream programming community due to its encapsulation of related data and functions into a single software component, its support for specialization and reuse, and its use as a clean mechanism for defining and implementing interfaces. Chapel supports objects in order to make these benefits available in a parallel language setting, and to provide a familiar paradigm for members of the mainstream programming community. Chapel supports traditional reference-based classes as well as value classes that are assigned and passed by value.

Chapel does not require the programmer to use an object-oriented style in their code, so that traditional Fortran and C programmers in the HPC community need not adopt a new programming paradigm in order to use Chapel effectively. Many of Chapel's standard library capabilities are implemented using objects, so such programmers may need to utilize a method-invocation style of syntax to use these capabilities. However, using such libraries does not necessitate broader adoption of object-oriented methodologies.

5.1.4 Generic Programming

Chapel's fourth principle is support for generic programming and polymorphism. These features allow code to be written in a style that is generic across types, making it applicable to variables of multiple types, sizes, and precisions. The goal of these features is to support exploratory programming as in popular interpreted and scripting languages, and to support code reuse by allowing algorithms to be expressed without explicitly replicating them for each possible type. This flexibility at the source level is implemented by having the compiler create versions of the code for each required type signature rather than by relying on dynamic typing which would result in unacceptable runtime overheads for the HPC community.

5.2 Getting Started

A Chapel version of the standard "hello, world" computation is as follows:

```
writeln("hello, world");
```

This complete Chapel program contains a single line of code that makes a call to the standard `writeln` function.

In general, Chapel programs define code using one or more named *modules*, each of which supports top-level initialization code that is invoked the first time the module is used. Programs also define a single entry point via a function named `main`. To facilitate exploratory programming, Chapel allows programmers to define modules using files rather than an explicit module declaration and to omit the program entry point when the program only has a single user module.

Chapel code is stored in files with the extension `.chpl`. Assuming the "hello, world" program is stored in a file called `hello.chpl`, it would define a single user module, `hello`, whose name is taken from the filename. Since the file defines a module, the top-level code in the file defines the module's initialization code. And since the program is composed of the single `hello` module, the `main` function is omitted. Thus, when the program is executed, the single `hello` module will be initialized by executing its top-level code thus invoking the call to the `writeln` function. Modules are described in more detail in §12.

To compile and run the "hello world" program, execute the following commands at the system prompt:

```
> chpl hello.chpl
> ./a.out
```

The following output will be printed to the console:

```
hello, world
```

6 Lexical Structure

This section describes the lexical components of Chapel programs. Note that the productions in this section are lexical; the components are not delimited by white space.

6.1 Comments

Two forms of comments are supported. All text following the consecutive characters `//` and before the end of the line is in a comment. All text following the consecutive characters `/*` and before the consecutive characters `*/` is in a comment.

Comments, including the characters that delimit them, do not affect the behavior of the program (except in delimiting tokens). If the delimiters that start the comments appear within a string literal, they do not start a comment but rather are part of the string literal.

Example. The following program makes use of both forms of comment:

```
/*
 *  main function
 */
def main() {
  writeln("hello, world"); // output greeting with new line
}
```

6.2 White Space

White-space characters are spaces, tabs, line feeds, and carriage returns. Along with comments, they delimit tokens, but are otherwise ignored.

6.3 Case Sensitivity

Chapel is a case sensitive language.

Example. The following identifiers are considered distinct: `chapel`, `Chapel`, and `CHAPEL`.

6.4 Tokens

Tokens include identifiers, keywords, literals, operators, and punctuation.

6.4.1 Identifiers

An identifier in Chapel is a sequence of characters that starts with a lowercase or uppercase letter or an underscore and is optionally followed by a sequence of lowercase or uppercase letters, digits, underscores, and dollar-signs. Identifiers are designated by the following syntax:

identifier:
  letter legal−identifier−chars$_{opt}$
  _ *legal−identifier−chars*

legal−identifier−chars:
  legal−identifier−char legal−identifier−chars$_{opt}$

legal−identifier−char:
  letter
  digit
  $

letter: *one of*
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

digit: *one of*
  0 1 2 3 4 5 6 7 8 9

Rationale. Why include “$” in the language? The inclusion of the $ character is meant to assist programmers using sync and single variables by supporting a convention (a $ at the end of such variables) in order to help write properly synchronized code. It is felt that marking such variables is useful since using such variables could result in deadlocks.

Example. The following are legal identifiers: `Cray1`, `syncvar$`, `legalIdentifier`, and `legal_identifier`.

6.4.2 Keywords

The following identifiers are reserved as keywords:

atomic	**delete**	**index**	**param**	**then**
begin	dmapped	**inout**	**record**	**true**
break	**do**	**label**	**reduce**	**type**
by	**domain**	lambda	**return**	**union**
class	**else**	**let**	**scan**	**use**
cobegin	**enum**	**local**	**select**	**var**
coforall	**false**	**module**	**serial**	**when**
config	**for**	**new**	**single**	**where**
const	**forall**	**nil**	**sparse**	**while**
continue	**if**	**on**	**subdomain**	**yield**
def	**in**	**otherwise**	**sync**	
		out		

6.4.3 Literals

Bool literals are designated by the following syntax:

> *bool−literal*: *one of*
>   **true false**

Signed and unsigned integer literals are designated by the following syntax:

> *integer−literal*:
>   *digits*
>   **0x** *hexadecimal−digits*
>   **0X** *hexadecimal−digits*
>   **0b** *binary−digits*
>   **0B** *binary−digits*
>
> *digits*:
>   *digit*
>   *digit digits*
>
> *hexadecimal−digits*:
>   *hexadecimal−digit*
>   *hexadecimal−digit hexadecimal−digits*
>
> *hexadecimal−digit*: *one of*
>   **0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f**
>
> *binary−digits*:
>   *binary−digit*
>   *binary−digit binary−digits*
>
> *binary−digit*: *one of*
>   **0 1**

Rationale. Why are there no suffixes on integral literals? Suffixes, like those in C, are not necessary. The type of an integer literal is the first type of the following that can hold the value of the digits: `int`, `int(64)`, `uint(64)`. Explicit conversions can then be used to change the type of the literal to another integer size.

Real literals are designated by the following syntax:

> *real−literal*:
>   *digits*$_{opt}$. *digits exponent−part*$_{opt}$
>   *digits exponent−part*
>
> *exponent−part*:
>   **e** *sign*$_{opt}$ *digits*
>   **E** *sign*$_{opt}$ *digits*
>
> *sign*: *one of*
>   + −

> *Rationale*. Why can't a real literal end with '.'? There is an ambiquity between real literals ending in '.' and the range operator '..' that makes it difficult to parse. For example, we want to parse `1..10` as a range from 1 to 10 without concern that `1.` is a real literal. There is also an ambiquity between methods invoked on literals. For example, if there is a method named `e` defined on integers, than `1.e` should invoke that method.

The type of a real literal is `real`. Explicit conversions are necessary to change the size of the literal.

Imaginary literals are designated by the following syntax:

imaginary−literal:
  real−literal **i**
  integer−literal **i**

The type of an imaginary literal is `imag`. Explicit conversions are necessary to change the size of the literal.

There are no complex literals. Rather, a complex value can be specified by adding or subtracting a real literal with an imaginary literal. Alternatively, a 2-tuple of integral or real expressions can be cast to a complex such that the first component becomes the real part and the second component becomes the imaginary part.

> *Example*. The following expressions are identical: `1.0 + 2.0i` and `(1.0, 2.0):complex`.

String literals are designated by the following syntax:

string−literal:
  " *double−quote−delimited−characters*$_{opt}$ "
  ' *single−quote−delimited−characters*$_{opt}$ '

double−quote−delimited−characters:
  string−character double−quote−delimited−characters$_{opt}$
  ' *double−quote−delimited−characters*$_{opt}$

single−quote−delimited−characters:
  string−character single−quote−delimited−characters$_{opt}$
  " *single−quote−delimited−characters*$_{opt}$

string−character:
  any character except the double quote, single quote, or new line
  simple−escape−character
  hexadecimal−escape−character

simple−escape−character: *one of*
  \\' \\" \\? \\a \\b \\f \\n \\r \\t \\v

hexadecimal−escape−character:
  \\x *hexadecimal−digits*

6.4.4 Operators and Punctuation

The following operators and punctuation are defined in the syntax of the language:

symbols	**use**
=	assignment
+= -= *= /= **= %= &= |= ^= &&= ||= <<= >>=	compound assignment
<=>	swap
..	range specifier
by	range/domain stride specifier
#	range count operator
...	variable argument lists
&& || ! & | ^ ~ << >>	logical/bitwise operators
== != <= >= < >	relational operators
+ - * / % **	arithmetic operators
:	type specifier
;	statement separator
,	expression separator
.	member access
?	type query
" '	string delimiters

6.4.5 Grouping Tokens

The following braces are part of the Chapel language:

braces	**use**
( )	parenthesization, function calls, and tuples
[ ]	domains, forall expressions, function calls, and tuples
{ }	compound statements

7 Types

Chapel is a statically typed language with a rich set of types. These include a set of predefined primitive types, enumerated types, locality types, structured types (classes, records, unions, tuples), data parallel types (ranges, domains, arrays), and synchronization types (sync, single).

The syntax of a type is summarized by the following syntax:

```
type-specifier:
  primitive-type
  enum-type
  locality-type
  structured-type
  dataparallel-type
  synchronization-type
```

The structured types are summarized by the following syntax:

```
structured-type:
  class-type
  record-type
  union-type
  tuple-type
```

Classes are discussed in §15. Records are discussed in §16. Unions are discussed in §17. Tuples are discussed in §14.

The data parallel types are summarized by the following syntax:

```
dataparallel-type:
  range-type
  domain-type
  mapped-domain-type
  array-type
  index-type
```

Ranges are discussed in §18. Domains, arrays, and index types are discussed in §19 and §20.

The synchronization types are summarized by the following syntax:

```
synchronization-type:
  sync-type
  single-type
```

Sync and single types are discussed in §24.1.2 and §24.1.3.

Programmers can define their own enumerated types, classes, records, unions, and type aliases in type declaration statements summarized by the following syntax:

```
type-declaration-statement:
  enum-declaration-statement
  class-declaration-statement
  record-declaration-statement
  union-declaration-statement
  type-alias-declaration-statement
```

7.1 Primitive Types

The primitive types include the following types: `bool`, `int`, `uint`, `real`, `imag`, `complex`, `string`, and `locale`. These primitive types are defined in this section.

The primitive types are summarized by the following syntax:

primitive-type:
  bool *primitive-type-parameter-part*$_{opt}$
  int *primitive-type-parameter-part*$_{opt}$
  uint *primitive-type-parameter-part*$_{opt}$
  real *primitive-type-parameter-part*$_{opt}$
  imag *primitive-type-parameter-part*$_{opt}$
  complex *primitive-type-parameter-part*$_{opt}$
  string

primitive-type-parameter-part:
  (*integer-parameter-expression*)

integer-parameter-expression:
  expression

Open issue. There is an expectation of future support for larger bit width primitive types depending on a platform's native support for those types.

7.1.1 The Bool Type

Chapel defines a logical data type designated by the symbol `bool` with the two predefined values `true` and `false`. This default boolean type is stored using an implementation-dependent number of bits. A particular number of bits can be specified using a parameter value following the `bool` keyword, such as `bool(8)` to request an 8-bit boolean value. Legal sizes are 8, 16, 32, and 64 bits.

Some statements require expressions of `bool` type and Chapel supports a special conversion of values to `bool` type when used in this context (§9.1.6).

7.1.2 Signed and Unsigned Integral Types

The integral types can be parameterized by the number of bits used to represent them. The default signed integral type, `int`, and the default unsigned integral type, `uint`, are 32 bits.

The integral types and their ranges are given in the following table:

Type	Minimum Value	Maximum Value
`int(8)`	-128	127
`uint(8)`	0	255
`int(16)`	-32768	32767
`uint(16)`	0	65535
`int(32),int`	-2147483648	2147483647
`uint(32),uint`	0	4294967295
`int(64)`	-9223372036854775808	9223372036854775807
`uint(64)`	0	18446744073709551615

The unary and binary operators that are pre-defined over the integral types operate with 32- and 64-bit precision. Using these operators on integral types represented with fewer bits results in a coercion according to the rules defined in §9.1.

Open issue. There is on going discussion on whether the default size of the integral types should be changed to 64 bits.

7.1.3 Real Types

Like the integral types, the real types can be parameterized by the number of bits used to represent them. The default real type, `real`, is 64 bits. The real types that are supported are machine-dependent, but usually include `real(32)` (single precision) and `real(64)` (double precision) following the IEEE 754 standard.

7.1.4 Imaginary Types

The imaginary types can be parameterized by the number of bits used to represent them. The default imaginary type, `imag`, is 64 bits. The imaginary types that are supported are machine-dependent, but usually include `imag(32)` and `imag(64).`

Rationale. The imaginary type is included to avoid numeric instabilities and under-optimized code stemming from always coercing real values to complex values with a zero imaginary part.

7.1.5 Complex Types

Like the integral and real types, the complex types can be parameterized by the number of bits used to represent them. A complex number is composed of two real numbers so the number of bits used to represent a complex is twice the number of bits used to represent the real numbers. The default complex type, `complex`, is 128 bits; it consists of two 64-bit real numbers. The complex types that are supported are machine-dependent, but usually include `complex(64)` and `complex(128).`

The real and imaginary components can be accessed via the methods `re` and `im`. The type of these components is real. See §30.1.1 for math routines for complex types.

Example. Given a complex number `c` with the value `3.14+2.72i`, the expressions `c.re` and `c.im` refer to `3.14` and `2.72` respectively.

7.1.6 The String Type

Strings are a primitive type designated by the symbol `string` comprised of ASCII characters. Their length is unbounded. See §30.1.2 for routines for manipulating strings.

> *Open issue*. There is an expectation of future support for fixed-length strings.

> *Open issue*. There is an expectation of future support for different character sets, possibly including internationalization.

7.2 Enumerated Types

Enumerated types are declared with the following syntax:

enum−declaration−statement:
  enum *identifier* { *enum−constant−list* } ;

enum−constant−list:
  enum−constant
  enum−constant , *enum−constant−list*

enum−constant:
  identifier init−part$_{opt}$

init−part:
  = *expression*

The enumerated type can then be specified with its name as summarized by the following syntax:

enum−type:
  identifier

An enumerated type defines a set of named constants that can be specified in a program as a member access on the enumerated type. These are associated with parameters of integral type. Each enumerated type is a distinct type. If the *init−part* is omitted, the *enum−constant* has an integral value one higher than the previous *enum−constant* in the enum, with the first having the value `1`.

Example. The code

```
enum statesman { Aristotle, Roosevelt, Churchill, Kissinger } ;
```

defines an enumerated type with four constants. The function

```
def quote(s: statesman) {
  select s {
    when statesman.Aristotle do
       writeln("All paid jobs absorb and degrade the mind.");
    when statesman.Roosevelt do
       writeln("Every reform movement has a lunatic fringe.");
    when statesman.Churchill do
       writeln("A joke is a very serious thing.");
```

```
          when statesman.Kissinger do
            { write("No one will ever win the battle of the sexes; ");
              writeln("there's too much fraternizing with the enemy."); }
      }
    }
```

outputs a quote from the given statesman. Note that enumerated constants must be prefixed by the enumerated type and a dot.

7.3 Locality Types

Locale types are summarized by the following syntax:

locality-type:
  locale

The `locale` type is defined in §26.1.1.

Open issue. We expect to support *realms* as another locality type.

7.4 Structured Types

7.4.1 Class Types

The class type defines a type that contains variables and constants, called fields, and functions, called methods. Classes are defined in §15. The class type can also contain type aliases and parameters. Such a class is generic and is defined in §22.3.

7.4.2 Record Types

The record type is similar to a class type; the primary difference is that a record is a value rather than a reference. Records are defined in §16.

7.4.3 Union Types

The union type defines a type that contains one of a set of variables. Like classes and records, unions may also define methods. Unions are defined in §17.

7.4.4 Tuple Types

A tuple is a light-weight record that consists of one or more anonymous fields. If all the fields are of the same type, the tuple is homogeneous. Tuples are defined in §14.

7.5 Data Parallel Types

7.5.1 Range Types

A range defines an integral sequence of some integral type. Ranges are defined in §18.

7.5.2 Domain, Array, and Index Types

A domain defines a set of indices and an array defines a set of elements that are mapped by the indices in an associated domain. A domain's indicies can be of any type. Domains, arrays, and index types are defined in §19 and §20.

7.6 Type Aliases

Type aliases are declared with the following syntax:

type−alias−declaration−statement:
  config$_{opt}$ **type** *type−alias−declaration−list* ;

type−alias−declaration−list:
  type−alias−declaration
  type−alias−declaration , *type−alias−declaration−list*

type−alias−declaration:
  identifier = *type−specifier*
  identifier

A type alias is a symbol that aliases the type specified in the *type−part*. A use of a type alias has the same meaning as using the type specified by *type−part* directly.

If the keyword `config` precedes the keyword `type`, the type alias is called a configuration type alias. Configuration type aliases are set at compilation time via compilation flags or other implementation dependent means. The `type-specifier` in the program is ignored if the type-alias is alternatively set.

The *type−part* is optional in the definition of a class or record. Such a type alias is called an unspecified type alias. Classes and records that contain type aliases, specified or unspecified, are generic (§22.3.1).

Open issue. There is on going discussion on whether a type alias is a new type or simply an alias. The former should enable redefinition of default values, identity elements, etc.

8 Variables

A variable is a symbol that represents memory. Chapel is a statically-typed, type-safe language so every variable has a type that is known at compile-time and the compiler enforces that values assigned to the variable can be stored in that variable as specified by its type.

8.1 Variable Declarations

Variables are declared with the following syntax:

variable-declaration-statement:
  config$_{opt}$ *variable-kind variable-declaration-list* ;

variable-kind: *one of*
  param const var

variable-declaration-list:
  variable-declaration
  variable-declaration-list , *variable-declaration*

variable-declaration:
  identifier-list type-part$_{opt}$ *initialization-part*
  identifier-list type-part
  array-alias-declaration

type-part:
  : *type-specifier*

initialization-part:
  = *expression*

identifier-list:
  identifier-list , *tuple-grouped-identifier-list*
  identifier-list , *identifier*
  tuple-grouped-identifier-list
  identifier

tuple-grouped-identifier-list:
  (*identifier-list*)

A *variable-declaration-statement* is used to define one or more variables. If the statement is a top-level module statement, the variables are global; otherwise they are local. Global variables are discussed in §8.2. Local variables are discussed in §8.3.

The optional keyword `config` specifies that the variables are configuration variables, described in Section §8.5.

The *variable-kind* specifies whether the variables are parameters (`param`), constants (`const`), or regular variables (`var`). Parameters are compile-time constants whereas constants are runtime constants. Both levels of constants are discussed in §8.4.

The *type-part* of a variable declaration specifies the type of the variable. It is optional if the *initialization-part* is specified. If the *type-part* is omitted, the type of the variable is inferred using local type inference described in §8.1.2.

The *initialization-part* of a variable declaration specifies an initial expression to assign to the variable. If the *initialization-part* is omitted, the variable is initialized to a default value described in §8.1.1.

Multiple variables can be defined in the same *variable-declaration-list*. The semantics of declaring multiple variables that share a *initialization-part* and/or *type-part* is defined in §8.1.3.

Multiple variables can be grouped together using a tuple notation as described in §14.4.2.

The *array-alias-declaration* is defined in §20.7.3.

8.1.1 Default Initialization

If a variable declaration has no initialization expression, a variable is initialized to the default value of its type. The default values are as follows:

Type	**Default Value**
`bool(*)`	`false`
`int(*)`	`0`
`uint(*)`	`0`
`real(*)`	`0.0`
`imag(*)`	`0.0i`
`complex(*)`	`0.0 + 0.0i`
`string`	`""`
enums	first enum constant
classes	`nil`
records	default constructed record
ranges	`1..0`
arrays	elements are default values
tuples	components are default values
sync/single	base default value and *empty* status

Open issue. Array initialization is potentially time-consuming. There is an expectation that there will be a way to declare an array that is explicitly left uninitialized in order to address this concern.

8.1.2 Local Type Inference

If the type is omitted from a variable declaration, the type of the variable is defined to be the type of the initialization expression. With the exception of sync and single expressions, the declaration

```
var v = e;
```

is equivalent to

```
var v: e.type;
v = e;
```

for an arbitrary expression `e`. For expressions of sync or single type, this translation does not hold because the evaluation of `e` results in a default read of this expression. The type of the variable is thus equal to the base type of the sync or single expression.

8.1.3 Multiple Variable Declarations

All variables defined in the same *identifier-list* are defined such that they have the same type and value, and so that the type and initialization expression are evaluated only once.

Example. In the declaration

```
def g() { writeln("side effect"); return "a string"; }
var a, b = 1.0, c, d:int, e, f = g();
```

variables `a` and `b` are of type `real` with value `1.0`. Variables `c` and `d` are of type `int` and are initialized to the default value of `0`. Variables `e` and `f` are of type `string` with value `"a string"`. The string `"side effect"` has been written to the display once. It is not evaluated twice.

The exact way that multiple variables are declared is defined as follows:

- If the variables in the *identifier-list* are declared with a type, but without an initialization expression as in

  ```
  var v1, v2, v3: t;
  ```

  for an arbitrary type expression `t`, then the declarations are rewritten so that the first variable is declared to be of type `t` and each later variable is declared to be of the type of the first variable as in

  ```
  var v1: t; var v2: v1.type; var v3: v1.type;
  ```

- If the variables in the *identifier-list* are declared without a type, but with an initialization expression as in

  ```
  var v1, v2, v3 = e;
  ```

  for an arbitrary expression `e`, then the declarations are rewritten so that the first variable is initialized by expression `e` and each later variable is initialized by the first variable as in

  ```
  var v1 = e; var v2 = v1; var v3 = v1;
  ```

- If the variables in the *identifier-list* are declared with both a type and an initialization expression as in

  ```
  var v1, v2, v3: t = e;
  ```

  for an arbitrary type expression `t` and an arbitrary expression `e`, then the declarations are rewritten so that the first variable is declared to be of type `t` and initialized by expression `e`, and each later variable is declared to be of the type of the first variable and initialized by the result of calling the function `readXX` on the first variable as in

  ```
  var v1: t = e; var v2: v1.type = readXX(v1); var v3: v1.type = readXX(v1);
  ```

where the function `readXX` is defined as follows:

```
def readXX(x: sync) return x.readXX();
def readXX(x: single) return x.readXX();
def readXX(x) return x;
```

Note that the `readXX` function cannot be called directly by a Chapel programmer and that adding an additional overload or shadowing this function will not change the semantics of variable declarations. This function is defined solely for the purposes of this specification.

Rationale. This algorithm is complicated by the existence of *sync* and *single* variables. If these did not exist, we could rewrite any multi-variable declaration such that later variables were simply initialized by the first variable and the first variable was defined as if it appeared alone in the *identifier-list*. However, both *sync* and *single* variables require careful handling to avoid unintentional changes to their *full/empty* state.

8.2 Global Variables

Variables declared in statements that are in a module but not in a function or block within that module are global variables. Global variables can be accessed anywhere within that module after the declaration of that variable. They can also be accessed in other modules that use that module.

8.3 Local Variables

Local variables are variables that are not global. Local variables are declared within block statements. They can only be accessed within the scope of that block statement (including all inner nested block statements and functions).

A local variable only exists during the execution of code that lies within that block statement. This time is called the lifetime of the variable. When execution has finished within that block statement, the local variable and the storage it represents is removed. Variables of class type are the sole exception. Constructors of class types create storage that is not associated with any scope. Such storage can be reclaimed as described in §15.11.

8.4 Constants

Constants are divided into two categories: parameters, specified with the keyword `param`, are compile-time constants and constants, specified with the keyword `const`, are runtime constants.

8.4.1 Compile-Time Constants

A compile-time constant, or "parameter", must have a single value that is known statically by the compiler. Parameters are restricted to primitive and enumerated types.

Parameters can be assigned expressions that are parameter expressions. Parameter expressions are restricted to the following constructs:

- Literals of primitive or enumerated type.
- Parenthesized parameter expressions.
- Casts of parameter expressions to primitive or enumerated types.
- Applications of the unary operators `+`, `-`, `!`, and `~` on operands that are bool or integral parameter expressions.
- Applications of the binary operators `+`, `-`, `*`, `/`, `%`, `**`, `&&`, `||`, `&`, `|`, `^`, <<, >>, ==, `!=`, <=, >=, <, and > on operands that are bool or integral parameter expressions.
- Applications of the string concatenation operator +, string comparison operators ==, `!=`, <=, >=, <, >, and the string length and ascii functions on parameter string expressions.
- The conditional expression where the condition is a parameter and the then- and else-expressions are parameters.
- Call expressions of parameter functions. See §13.8.

There is an expectation that parameters will be expanded to more types and more operations.

8.4.2 Runtime Constants

Runtime constants, or simply "constants", do not have the restrictions that are associated with parameters. Constants can be any type. They require an initialization expression and contain the value of that expression throughout their lifetime.

A variable of a class type that is a constant is a constant reference. That is, the variable always points to the object that it was initialized to reference. However, the fields of that object are allowed to be modified.

8.5 Configuration Variables

If the keyword `config` precedes the keyword `var`, `const`, or `param`, the variable, constant, or parameter is called a configuration variable, configuration constant, or configuration parameter respectively. Such variables, constants, and parameters must be global.

The initialization of these variables can be set via implementation dependent means, such as command-line switches or environment variables. The initialization expression in the program is ignored if the initialization is alternatively set.

Configuration parameters are set at compilation time via compilation flags or other implementation dependent means. The value passed via these means can be an arbitrary Chapel expression as long as the expression can be evaluated at compile-time.

Example. A configuration parameter can be set via a compiler flag. For example,

```
config param rank = 2;
```

sets a integer parameter `rank` to `2`. At compile-time, `rank` can be set via to `3` or `2*n` (assuming `n` is also a param variable) or any other expression that can be evaluated at compile-time. This can be used to write rank-independent code.

9 Conversions

A conversion allows an expression of one type to be converted into another type. Conversions can be either implicit (§9.1) or explicit (§9.2).

9.1 Implicit Conversions

Implicit conversions can occur during an assignment (from the expression on the right-hand side to the variable on the left-hand side) or during a function call (from the actual expression to the formal argument). An implicit conversion does not require a cast.

Implicit conversions are allowed between numeric types (§9.1.1), from enumerated types to numeric types (§9.1.2), between class types (§9.1.3), and between record types (§9.1.4). A special set of implicit conversions are allowed from compile-time constants of type `int` and `int(64)` to other smaller numeric types if the value is in the range of the smaller numeric type (§9.1.5). Lastly, implicit conversions are supported from integral and class types to bool in the context of a statement (§9.1.6).

9.1.1 Implicit Bool and Numeric Conversions

The implicit numeric conversions are as follows:

- From `bool` to `bool(k)`, `int(8)`, `int(16)`, `int(32)`, `int(64)`, `uint(8)`, `uint(16)`, `uint(32)`, or `uint(64)` for any legal value of k
- From `bool(j)` to `bool`, `bool(k)`, `int(8)`, `int(16)`, `int(32)`, `int(64)`, `uint(8)`, `uint(16)`, `uint(32)`, or `uint(64)`, for any legal values of j and k
- From `int(8)` to `int(16)`, `int(32)`, `int(64)`, `real(64)`, or `complex(128)`
- From `int(16)` to `int(32)`, `int(64)`, `real(64)`, or `complex(128)`
- From `int(32)` to `int(64)`, `real(64)`, or `complex(128)`
- From `int(64)` to `real(64)`, or `complex(128)`
- From `uint(8)` to `int(16)`, `int(32)`, `int(64)`, `uint(16)`, `uint(32)`, `uint(64)`, `real(64)`, or `complex(128)`
- From `uint(16)` to `int(32)`, `int(64)`, `uint(32)`, `uint(64)`, `real(64)`, or `complex(128)`
- From `uint(32)` to `int(64)`, `uint(64)`, `real(64)`, or `complex(128)`
- From `uint(64)` to `real(64)`, or `complex(128)`
- From `real(32)` to `real(64)`, `complex(64)`, or `complex(128)`
- From `real(64)` to `complex(128)`
- From `imag(32)` to `imag(64)`, `complex(64)`, or `complex(128)`
- From `imag(64)` to `complex(128)`

- From `complex(64)` to `complex(128)`

The implicit numeric conversions do not result in any loss of information except for the conversions from `int(64)` or `uint(64)` to `real(64)` or `complex(128)`.

> *Rationale*. In C#, implicit conversions from `int(32)` or `int(64)` to `real(32)` are supported and allow for a loss of precision. Since the default `real` size is 64 and the default `int` size is 32 in Chapel, we did not follow the lead of C# in this regard since it seemed unfortunate to favor `real(32)` over `real` in the default case. That is, given the `sqrt` function defined over `real(32)` and `real`, it is preferable to choose the version over `real` when calling with an actual of type `int` rather than lose precision and half of the bits to call the `real(32)` version.
>
> Additionally, we don't allow implicit conversions from `int(8)` or `int(16)` to `real(32)` because to do so would result in an ambiguity when computing, e.g., `int(8) + int(8)`.

9.1.2 Implicit Enumeration Conversions

An expression that is an enumerated type can be implicitly converted to any integral type as long as all of the constants defined by the enumerated type are within range of the integral type.

9.1.3 Implicit Class Conversions

An expression of class type `D` can be implicitly converted to another class type `C` provided that `D` is a subclass of `C`.

9.1.4 Implicit Record Conversions

An expression of record type `D` can be implicitly converted to another record type `C` provided that `D` is a nominal subtype of `C`.

9.1.5 Implicit Compile-Time Constant Conversions

The following two implicit conversions of parameters are supported:

- A parameter of type `int(32)` can be implicitly converted to `int(8)`, `int(16)`, or any unsigned integral type if the value of the parameter is within the range of the target type.
- A parameter of type `int(64)` can be implicitly converted to `uint(64)` if the value of the parameter is nonnegative.

9.1.6 Implicit Statement Bool Conversions

In the condition of an if-statement, while-loop, and do-while-loop, the following implicit conversions are supported:

- An expression of integral type is taken to be true if it is non-zero and is false otherwise.
- An expression of a class type is taken to be true if it is not nil and is false otherwise.

9.2 Explicit Conversions

Explicit conversions require a cast in the code. Casts are defined in §10.9. Explicit conversions are supported between more types than implicit conversions, but explicit conversions are not supported between all types.

The explicit conversions are a superset of the implicit conversions.

9.2.1 Explicit Numeric Conversions

Explicit conversions are allowed from any numeric type, bool, or string to any other numeric type, bool, or string. The definitions of how these explicit conversions work is forthcoming.

9.2.2 Explicit Enumeration Conversions

Explicit conversions are allowed from any enumerated types to any numeric type, bool, or string, and vice versa.

9.2.3 Explicit Class Conversions

An expression of static class type `C` can be explicitly converted to a class type `D` provided that `C` is derived from `D` or `D` is derived from `C`. In the event that `D` is derived from `C`, it is a runtime error if the the dynamic class type of `C` is not derived from or equal to `D`.

9.2.4 Explicit Record Conversions

An expression of record type `C` can be explicitly converted to another record type `D` provided that `C` is derived from `D`. There are no explicit record conversions that are not also implicit record conversions.

10 Expressions

This section defines expressions in Chapel with the following exceptions: The forall expression, reduce expressions, and scan expressions are defined in §25. Module access expressions are defined in §12.4.1. Tuple expressions and tuple expand expressions are defined in §14. Locale access expressions are defined in §26.1.5. Mapped domain expressions are defined in §27.

The syntax for an expression is given by:

```
expression:
  literal-expression
  variable-expression
  enum-constant-expression
  member-access-expression
  call-expression
  query-expression
  cast-expression
  lvalue-expression
  parenthesized-expression
  unary-expression
  binary-expression
  let-expression
  if-expression
  for-expression
  forall-expression
  reduce-expression
  scan-expression
  module-access-expression
  tuple-expression
  tuple-expand-expression
  locale-access-expression
  mapped-domain-expression
```

10.1 Literal Expressions

A literal value for any of the built-in types (§6.4.3) is a literal expression. Literal expressions are given by the following syntax:

```
literal-expression:
  bool-literal
  integer-literal
  real-literal
  imaginary-literal
  string-literal
  range-literal
  domain-literal
```

10.2 Variable Expressions

A use of a variable, constant, parameter, or formal argument, is itself an expression. The syntax of a variable expression is given by:

variable−expression:
  identifier

10.3 Enumeration Constant Expression

A use of an enumeration constant is itself an expression. Such a constant must be preceded by the enumeration type name. The syntax of an enumeration constant expression is given by:

enum−constant−expression:
  enum−type . identifier

Example. For an example of using enumeration constants, see §7.2.

10.4 Parenthesized Expressions

A *parenthesized−expression* is an expression that is delimited by parentheses as given by:

parenthesized−expression:
  (*expression*)

Such an expression evaluates to the expression. The parentheses is ignored and has only syntactic effect.

10.5 Call Expressions

Functions and function calls are defined in §13.

10.6 Indexing Expressions

Indexing into arrays, tuples, and domains shares the same syntax of a call expression. Indexing, at its core, is nothing more than a call to the indexing function defined on these types.

10.7 Member Access Expressions

Member access expressions are call expressions to members of classes, records, or unions. The syntax for a member access is given by:

member−access−expression:
  expression . identifier

The member access may be an access of a field or a function inside a class, record, or union.

10.8 The Query Expression

A query expression is used to query a type or value within a formal argument type expression. The syntax of a query expression is given by:

query−expression:
  ? *identifier*$_{opt}$

Querying is restricted to querying the type of a formal argument, the element type of a formal argument that is an array, the domain of a formal argument that is an array, the size of a primitive type, or a type or parameter field of a formal argument type.

The identifier can be omitted. This is useful for ensuring the genericity of a generic type that defines default values for all of its generic fields when specifying a formal argument as discussed in §22.1.5.

Example. The following code defines a generic function where the type of the first parameter is queried and stored in the type alias `t` and the domain of the second argument is queried and stored in the variable `D`:

```
def foo(x: ?t, y: [?D] t) {
  for i in D do
    y[i] = x;
}
```

This allows a generic specification of a function to assign a particular value to all elements of an array. The value and the elements of the array are constrained to be the same type. This function can be rewritten without query expression as follows:

```
def foo(x, y: [] x.type) {
  for i in y.domain do
    y[i] = x;
}
```

There is an expectation that query expressions will be allowed in more places in the future.

10.9 Casts

A cast is specified with the following syntax:

cast−expression:
  expression : *type−specifier*

The expression is converted to the specified type. Except for the casts listed below, casts are restricted to valid explicit conversions (§9.2).

The following cast has a special meaning and does not correspond to an explicit conversion:

- A cast from a 2-tuple to `complex` converts the 2-tuple into a complex where the first component becomes the real part and the second component becomes the imaginary part. The size of the complex is determined from the size of the components based on implicit conversions.

10.10 LValue Expressions

An *lvalue* is an expression that can be used on the left-hand side of an assignment statement or on either side of a swap statement, that can be passed to a formal argument of a function that has `out` or `inout` intent, or that can be returned by a variable function. Valid lvalue expressions include the following:

- Variable expressions.
- Member access expressions.
- Call expressions of variable functions.
- Indexing expressions.

LValue expressions are given by the following syntax:

```
lvalue-expression:
  variable-expression
  member-access-expression
  call-expression
```

The syntax is less restrictive than the definition above. For example, not all *call-expression*s are lvalues.

10.11 Precedence and Associativity

The following table summarizes operator and expression precedence and associativity. Operators and expressions listed earlier have higher precedence than those listed later.

Operator	**Associativity**	**Use**

`.`
`()`
`[]`
	left	member access
		function call or access (zipper)
		function call or access (tensor)

`new`	right	constructor call
`:`	left	cast
`**`	right	exponentiation

`reduce`
`scan`
`dmapped`
	left	reduction
		scan
		domain map application

`!` `~`	right	logical negation bitwise negation

`*`
`/`
`%`
	left	multiplication
		division
		modulus

unary + unary −	right	positive identity negation
`+` `-`	left	addition subtraction

`<<`
`>>`
	left	left shift
		right shift

`<=`
`>=`
`<`
`>`
	left	less-than-or-equal-to comparison
		greater-than-or-equal-to comparison
		less-than comparison
		greater-than comparison

`==`
`!=`
	left	equal-to comparison
		not-equal-to comparison

`&`	left	bitwise/logical and
`^`	left	bitwise/logical xor
`|`	left	bitwise/logical or
`&&`	left	short-circuiting logical and
`||`	left	short-circuiting logical or
`..`	left	range construction
`in`	left	forall expression
`by` `#`	left	range/domain stride application range count application

`if then else`
`forall do`
`[ ]`
`for do`
`sync single`
	left	conditional expression
		forall expression
		forall expression
		for expression
		sync and single type

`,`	left	comma separated expressions

Rationale. In general, our operator precedence is based on that of the C family of languages including C++, Java, Perl, and C#. We comment on a few of the differences and unique factors here.

We find that there is tension between the relative precedence of exponentiation, unary minus/plus, and casts. The following three expressions show our intuition for how these expressions should be parenthesized.

```
-2**4            wants   -(2**4)
-2:uint          wants   (-2):uint
2:uint**4:uint   wants   (2:uint)**(4:uint)
```

Trying to support all three of these cases results in a circularity—exponentiation wants precedence over unary minus, unary minus wants precedence over casts, and casts want precedence over exponentiation. We chose to break the circularity by making unary minus have a lower precedence. This means that for the second case above:

```
-2:uint   requires   (-2):uint
```

We also chose to depart from the C family of languages by making unary plus/minus have lower precedence than binary multiplication, division, and modulus as in Fortran. We have found very few cases that distinguish between these cases. An interesting one is:

```
const minint = min(int(32));
...-minint/2...
```

Intuitively, this should result in a positive value, yet C's precedence rules results in a negative value due to asymmetry in modern integer representations. If we learn of cases that argue in favor of the C approach, we would likely reverse this decision in order to more closely match C.

We were tempted to diverge from the C precedence rules for the binary bitwise operators to make them bind less tightly than comparisons. This would allow us to interpret:

```
a | b == 0   as   (a | b) == 0
```

However, given that no other popular modern language has made this change, we felt it unwise to stray from the pack. The typical rationale for the C ordering is to allow these operators to be used as non-short-circuiting logical operations.

One final area of note is the precedence of reductions. Two common cases tend to argue for making reductions very low or very high in the precedence table:

```
max reduce A - min reduce A   wants   (max reduce A) - (min reduce A)
max reduce A * B              wants   max reduce (A * B)
```

The first statement would require reductions to have a higher precedence than the arithmetic operators while the second would require them to be lower. We opted to make reductions have high precedence due to the argument that they tend to resemble unary operators. Thus, to support our intuition:

```
max reduce A * B   requires   max reduce (A * B)
```

This choice also has the (arguably positive) effect of making the unparenthesized version of this statement result in an aggregate value if A and B are both aggregates—the reduction of A results in a scalar which promotes when being multiplied by B, resulting in an aggregate. Our intuition is that users who forget the parenthesis will learn of their error at compilation time because the resulting expression is not a scalar as expected.

10.12 Operator Expressions

The application of operators to expressions is itself an expression. The syntax of a unary expression is given by:

unary−expression:
  unary−operator expression

unary−operator: *one of*
  + −˜ !

The syntax of a binary expression is given by:

binary−expression:
  expression binary−operator expression

binary−operator: *one of*
  + −∗ / % ∗∗ & | ˆ << >> && || == != <= >= < > **by** #

The operators are defined in subsequent sections.

10.13 Arithmetic Operators

This section describes the predefined arithmetic operators. These operators can be redefined over different types using operator overloading (§13.12).

All integral arithmetic operators are implemented over integral types of size 32 and 64 bits only. For example, adding two 8-bit integers is done by first converting them to 32-bit integers and then adding the 32-bit integers. The result is a 32-bit integer.

10.13.1 Unary Plus Operators

The unary plus operators are predefined as follows:

```
def +(a: int(32)): int(32)
def +(a: int(64)): int(64)
def +(a: uint(32)): uint(32)
def +(a: uint(64)): uint(64)
def +(a: real(32)): real(32)
def +(a: real(64)): real(64)
def +(a: imag(32)): imag(32)
def +(a: imag(64)): imag(64)
def +(a: complex(32)): complex(32)
def +(a: complex(64)): complex(64)
def +(a: complex(128)): complex(128)
```

For each of these definitions, the result is the value of the operand.

10.13.2 Unary Minus Operators

The unary minus operators are predefined as follows:

```
def -(a: int(32)): int(32)
def -(a: int(64)): int(64)
def -(a: uint(64))
def -(a: real(32)): real(32)
def -(a: real(64)): real(64)
def -(a: imag(32)): imag(32)
def -(a: imag(64)): imag(64)
def -(a: complex(32)): complex(32)
def -(a: complex(64)): complex(64)
def -(a: complex(128)): complex(128)
```

For each of these definitions that return a value, the result is the negation of the value of the operand. For integral types, this corresponds to subtracting the value from zero. For real and imaginary types, this corresponds to inverting the sign. For complex types, this corresponds to inverting the signs of both the real and imaginary parts.

It is an error to try to negate a value of type `uint(64)`. Note that negating a value of type `uint(32)` first converts the type to `int(64)` using an implicit conversion.

10.13.3 Addition Operators

The addition operators are predefined as follows:

```
def +(a: int(32), b: int(32)): int(32)
def +(a: int(64), b: int(64)): int(64)
def +(a: uint(32), b: uint(32)): uint(32)
def +(a: uint(64), b: uint(64)): uint(64)
def +(a: uint(64), b: int(64))
def +(a: int(64), b: uint(64))

def +(a: real(32), b: real(32)): real(32)
def +(a: real(64), b: real(64)): real(64)

def +(a: imag(32), b: imag(32)): imag(32)
def +(a: imag(64), b: imag(64)): imag(64)

def +(a: complex(64), b: complex(64)): complex(64)
def +(a: complex(128), b: complex(128)): complex(128)

def +(a: real(32), b: imag(32)): complex(64)
def +(a: imag(32), b: real(32)): complex(64)
def +(a: real(64), b: imag(64)): complex(128)
def +(a: imag(64), b: real(64)): complex(128)

def +(a: real(32), b: complex(64)): complex(64)
def +(a: complex(64), b: real(32)): complex(64)
def +(a: real(64), b: complex(128)): complex(128)
def +(a: complex(128), b: real(64)): complex(128)

def +(a: imag(32), b: complex(64)): complex(64)
def +(a: complex(64), b: imag(32)): complex(64)
def +(a: imag(64), b: complex(128)): complex(128)
def +(a: complex(128), b: imag(64)): complex(128)
```

For each of these definitions that return a value, the result is the sum of the two operands.

It is a compile-time error to add a value of type `uint(64)` and a value of type `int(64)`.

Addition over a value of real type and a value of imaginary type produces a value of complex type. Addition of values of complex type and either real or imaginary types also produces a value of complex type.

10.13.4 Subtraction Operators

The subtraction operators are predefined as follows:

```
def -(a: int(32), b: int(32)): int(32)
def -(a: int(64), b: int(64)): int(64)
def -(a: uint(32), b: uint(32)): uint(32)
def -(a: uint(64), b: uint(64)): uint(64)
def -(a: uint(64), b: int(64))
def -(a: int(64), b: uint(64))

def -(a: real(32), b: real(32)): real(32)
def -(a: real(64), b: real(64)): real(64)

def -(a: imag(32), b: imag(32)): imag(32)
def -(a: imag(64), b: imag(64)): imag(64)

def -(a: complex(64), b: complex(64)): complex(64)
def -(a: complex(128), b: complex(128)): complex(128)

def -(a: real(32), b: imag(32)): complex(64)
def -(a: imag(32), b: real(32)): complex(64)
def -(a: real(64), b: imag(64)): complex(128)
def -(a: imag(64), b: real(64)): complex(128)

def -(a: real(32), b: complex(64)): complex(64)
def -(a: complex(64), b: real(32)): complex(64)
def -(a: real(64), b: complex(128)): complex(128)
def -(a: complex(128), b: real(64)): complex(128)

def -(a: imag(32), b: complex(64)): complex(64)
def -(a: complex(64), b: imag(32)): complex(64)
def -(a: imag(64), b: complex(128)): complex(128)
def -(a: complex(128), b: imag(64)): complex(128)
```

For each of these definitions that return a value, the result is the value obtained by subtracting the second operand from the first operand.

It is a compile-time error to subtract a value of type `uint(64)` from a value of type `int(64)`, and vice versa.

Subtraction of a value of real type from a value of imaginary type, and vice versa, produces a value of complex type. Subtraction of values of complex type from either real or imaginary types, and vice versa, also produces a value of complex type.

10.13.5 Multiplication Operators

The multiplication operators are predefined as follows:

```
def *(a: int(32), b: int(32)): int(32)
def *(a: int(64), b: int(64)): int(64)
def *(a: uint(32), b: uint(32)): uint(32)
def *(a: uint(64), b: uint(64)): uint(64)
def *(a: uint(64), b: int(64))
def *(a: int(64), b: uint(64))

def *(a: real(32), b: real(32)): real(32)
def *(a: real(64), b: real(64)): real(64)

def *(a: imag(32), b: imag(32)): real(32)
def *(a: imag(64), b: imag(64)): real(64)

def *(a: complex(64), b: complex(64)): complex(64)
def *(a: complex(128), b: complex(128)): complex(128)

def *(a: real(32), b: imag(32)): imag(32)
def *(a: imag(32), b: real(32)): imag(32)
def *(a: real(64), b: imag(64)): imag(64)
def *(a: imag(64), b: real(64)): imag(64)

def *(a: real(32), b: complex(64)): complex(64)
def *(a: complex(64), b: real(32)): complex(64)
def *(a: real(64), b: complex(128)): complex(128)
def *(a: complex(128), b: real(64)): complex(128)

def *(a: imag(32), b: complex(64)): complex(64)
def *(a: complex(64), b: imag(32)): complex(64)
def *(a: imag(64), b: complex(128)): complex(128)
def *(a: complex(128), b: imag(64)): complex(128)
```

For each of these definitions that return a value, the result is the product of the two operands.

It is a compile-time error to multiply a value of type `uint(64)` and a value of type `int(64)`.

Multiplication of values of imaginary type produces a value of real type. Multiplication over a value of real type and a value of imaginary type produces a value of imaginary type. Multiplication of values of complex type and either real or imaginary types produces a value of complex type.

10.13.6 Division Operators

The division operators are predefined as follows:

```
def /(a: int(32), b: int(32)): int(32)
def /(a: int(64), b: int(64)): int(64)
def /(a: uint(32), b: uint(32)): uint(32)
def /(a: uint(64), b: uint(64)): uint(64)
def /(a: uint(64), b: int(64))
def /(a: int(64), b: uint(64))

def /(a: real(32), b: real(32)): real(32)
def /(a: real(64), b: real(64)): real(64)

def /(a: imag(32), b: imag(32)): real(32)
def /(a: imag(64), b: imag(64)): real(64)

def /(a: complex(64), b: complex(64)): complex(64)
def /(a: complex(128), b: complex(128)): complex(128)
def /(a: real(32), b: imag(32)): imag(32)
def /(a: imag(32), b: real(32)): imag(32)
def /(a: real(64), b: imag(64)): imag(64)
def /(a: imag(64), b: real(64)): imag(64)

def /(a: real(32), b: complex(64)): complex(64)
def /(a: complex(64), b: real(32)): complex(64)
def /(a: real(64), b: complex(128)): complex(128)
def /(a: complex(128), b: real(64)): complex(128)

def /(a: imag(32), b: complex(64)): complex(64)
def /(a: complex(64), b: imag(32)): complex(64)
def /(a: imag(64), b: complex(128)): complex(128)
def /(a: complex(128), b: imag(64)): complex(128)
```

For each of these definitions that return a value, the result is the quotient of the two operands.

It is a compile-time error to divide a value of type `uint(64)` by a value of type `int(64)`, and vice versa.

Division of values of imaginary type produces a value of real type. Division over a value of real type and a value of imaginary type produces a value of imaginary type. Division of values of complex type and either real or imaginary types produces a value of complex type.

10.13.7 Modulus Operators

The modulus operators are predefined as follows:

```
def %(a: int(32), b: int(32)): int(32)
def %(a: int(64), b: int(64)): int(64)
def %(a: uint(32), b: uint(32)): uint(32)
def %(a: uint(64), b: uint(64)): uint(64)
def %(a: uint(64), b: int(64))
def %(a: int(64), b: uint(64))
```

For each of these definitions that return a value, the result is the remainder when the first operand is divided by the second operand.

It is a compile-time error to take the remainder of a value of type `uint(64)` and a value of type `int(64)`, and vice versa.

There is an expectation that the predefined modulus operators will be extended to handle real, imaginary, and complex types in the future.

10.13.8 Exponentiation Operators

The exponentiation operators are predefined as follows:

```
def **(a: int(32), b: int(32)): int(32)
def **(a: int(64), b: int(64)): int(64)
def **(a: uint(32), b: uint(32)): uint(32)
def **(a: uint(64), b: uint(64)): uint(64)
def **(a: uint(64), b: int(64))
def **(a: int(64), b: uint(64))

def **(a: real(32), b: real(32)): real(32)
def **(a: real(64), b: real(64)): real(64)
```

For each of these definitions that return a value, the result is the value of the first operand raised to the power of the second operand.

It is a compile-time error to take the exponent of a value of type `uint(64)` by a value of type `int(64)`, and vice versa.

There is an expectation that the predefined exponentiation operators will be extended to handle imaginary and complex types in the future.

10.14 Bitwise Operators

This section describes the predefined bitwise operators. These operators can be redefined over different types using operator overloading (§13.12).

10.14.1 Bitwise Complement Operators

The bitwise complement operators are predefined as follows:

```
def ~(a: bool): bool
def ~(a: int(32)): int(32)
def ~(a: int(64)): int(64)
def ~(a: uint(32)): uint(32)
def ~(a: uint(64)): uint(64)
```

For each of these definitions, the result is the bitwise complement of the operand.

10.14.2 Bitwise And Operators

The bitwise and operators are predefined as follows:

```
def &(a: bool, b: bool): bool
def &(a: int(32), b: int(32)): int(32)
def &(a: int(64), b: int(64)): int(64)
def &(a: uint(32), b: uint(32)): uint(32)
def &(a: uint(64), b: uint(64)): uint(64)
def &(a: int(32), b: uint(32)): uint(32)
def &(a: int(64), b: uint(64)): uint(64)
def &(a: uint(32), b: int(32)): uint(32)
def &(a: uint(64), b: int(64)): uint(64)
```

For each of these definitions, the result is computed by applying the logical and operation to the bits of the operands.

Chapel allows mixing signed and unsigned integers of the same size when passing them as arguments to bitwise and. In the mixed case the result is of the same size as the arguments and is unsigned.

Rationale. The mathematical meaning of integer arguments is discarded when they are passed to bitwise operators. Instead the arguments are treated simply as bit vectors. The bit-vector meaning is preserved when converting between signed and unsigned of the same size. The choice of unsigned over signed as the result type in the mixed case reflects the semantics of standard C.

10.14.3 Bitwise Or Operators

The bitwise or operators are predefined as follows:

```
def |(a: bool, b: bool): bool
def |(a: int(32), b: int(32)): int(32)
def |(a: int(64), b: int(64)): int(64)
def |(a: uint(32), b: uint(32)): uint(32)
def |(a: uint(64), b: uint(64)): uint(64)
def |(a: int(32), b: uint(32)): uint(32)
def |(a: int(64), b: uint(64)): uint(64)
def |(a: uint(32), b: int(32)): uint(32)
def |(a: uint(64), b: int(64)): uint(64)
```

For each of these definitions, the result is computed by applying the logical or operation to the bits of the operands.

Chapel allows mixing signed and unsigned integers of the same size when passing them as arguments to bitwise or.

Rationale. The same as for bitwise and (§10.14.2).

10.14.4 Bitwise Xor Operators

The bitwise xor operators are predefined as follows:

```
def ^(a: bool, b: bool): bool
def ^(a: int(32), b: int(32)): int(32)
def ^(a: int(64), b: int(64)): int(64)
def ^(a: uint(32), b: uint(32)): uint(32)
def ^(a: uint(64), b: uint(64)): uint(64)
def ^(a: int(32), b: uint(32)): uint(32)
def ^(a: int(64), b: uint(64)): uint(64)
def ^(a: uint(32), b: int(32)): uint(32)
def ^(a: uint(64), b: int(64)): uint(64)
```

For each of these definitions, the result is computed by applying the XOR operation to the bits of the operands.

Chapel allows mixing signed and unsigned integers of the same size when passing them as arguments to bitwise xor.

Rationale. The same as for bitwise and (§10.14.2).

10.15 Shift Operators

This section describes the predefined shift operators. These operators can be redefined over different types using operator overloading (§13.12).

The shift operators are predefined as follows:

```
def <<(a: int(32), b): int(32)
def >>(a: int(32), b): int(32)
def <<(a: int(64), b): int(64)
def >>(a: int(64), b): int(64)
def <<(a: uint(32), b): uint(32)
def >>(a: uint(32), b): uint(32)
def <<(a: uint(64), b): uint(64)
def >>(a: uint(64), b): uint(64)
```

The type of the second actual argument must be any integral type.

The `<<` operator shifts the bits of `a` left by the integer `b`. The new low-order bits are set to zero.

The `>>` operator shifts the bits of `a` right by the integer `b`. When `a` is negative, the new high-order bits are set to one; otherwise the new high-order bits are set to zero.

The value of `b` must be non-negative.

10.16 Logical Operators

This section describes the predefined logical operators. These operators can be redefined over different types using operator overloading (§13.12).

10.16.1 The Logical Negation Operator

The logical negation operator is predefined as follows:

```
def !(a: bool): bool
```

The result is the logical negation of the operand.

10.16.2 The Logical And Operator

The logical and operator is predefined over bool type. It returns true if both operands evaluate to true; otherwise it returns false. If the first operand evaluates to false, the second operand is not evaluated and the result is false.

The logical and operator over expressions `a` and `b` given by

```
a && b
```

is evaluated as the expression

```
if isTrue(a) then isTrue(b) else false
```

The function `isTrue` is predefined over bool type as follows:

```
def isTrue(a:bool) return a;
```

Overloading the logical and operator over other types is accomplished by overloading the `isTrue` function over other types.

10.16.3 The Logical Or Operator

The logical or operator is predefined over bool type. It returns true if either operand evaluate to true; otherwise it returns false. If the first operand evaluates to true, the second operand is not evaluated and the result is true.

The logical or operator over expressions `a` and `b` given by

```
a || b
```

is evaluated as the expression

```
if isTrue(a) then true else isTrue(b)
```

The function `isTrue` is predefined over bool type as described in §10.16.2. Overloading the logical or operator over other types is accomplished by overloading the `isTrue` function over other types.

10.17 Relational Operators

This section describes the predefined relational operators. These operators can be redefined over different types using operator overloading (§13.12).

10.17.1 Ordered Comparison Operators

The "less than" comparison operators are predefined over numeric types as follows:

```
def <(a: int(32), b: int(32)): bool
def <(a: int(64), b: int(64)): bool
def <(a: uint(32), b: uint(32)): bool
def <(a: uint(64), b: uint(64)): bool
def <(a: real(32), b: real(32)): bool
def <(a: real(64), b: real(64)): bool
def <(a: imag(32), b: imag(32)): bool
def <(a: imag(64), b: imag(64)): bool
```

The result of `a < b` is true if `a` is less than `b`; otherwise the result is false.

The "greater than" comparison operators are predefined over numeric types as follows:

```
def >(a: int(32), b: int(32)): bool
def >(a: int(64), b: int(64)): bool
def >(a: uint(32), b: uint(32)): bool
def >(a: uint(64), b: uint(64)): bool
def >(a: real(32), b: real(32)): bool
def >(a: real(64), b: real(64)): bool
def >(a: imag(32), b: imag(32)): bool
def >(a: imag(64), b: imag(64)): bool
```

The result of `a > b` is true if `a` is greater than `b`; otherwise the result is false.

The "less than or equal to" comparison operators are predefined over numeric types as follows:

```
def <=(a: int(32), b: int(32)): bool
def <=(a: int(64), b: int(64)): bool
def <=(a: uint(32), b: uint(32)): bool
def <=(a: uint(64), b: uint(64)): bool
def <=(a: real(32), b: real(32)): bool
def <=(a: real(64), b: real(64)): bool
def <=(a: imag(32), b: imag(32)): bool
def <=(a: imag(64), b: imag(64)): bool
```

The result of `a <= b` is true if `a` is less than or equal to `b`; otherwise the result is false.

The "greater than or equal to" comparison operators are predefined over numeric types as follows:

```
def >=(a: int(32), b: int(32)): bool
def >=(a: int(64), b: int(64)): bool
def >=(a: uint(32), b: uint(32)): bool
def >=(a: uint(64), b: uint(64)): bool
def >=(a: real(32), b: real(32)): bool
def >=(a: real(64), b: real(64)): bool
def >=(a: imag(32), b: imag(32)): bool
def >=(a: imag(64), b: imag(64)): bool
```

The result of `a >= b` is true if `a` is greater than or equal to `b`; otherwise the result is false.

The ordered comparison operators are predefined over strings as follows:

```
def <(a: string, b: string): bool
def >(a: string, b: string): bool
def <=(a: string, b: string): bool
def >=(a: string, b: string): bool
```

Comparisons between strings are defined based on the ordering of the character set used to represent the string, which is applied elementwise to the string's characters in order.

10.17.2 Equality Comparison Operators

The equality comparison operators are predefined over bool and the numeric types as follows:

```
def ==(a: int(32), b: int(32)): bool
def ==(a: int(64), b: int(64)): bool
def ==(a: uint(32), b: uint(32)): bool
def ==(a: uint(64), b: uint(64)): bool
def ==(a: real(32), b: real(32)): bool
def ==(a: real(64), b: real(64)): bool
def ==(a: imag(32), b: imag(32)): bool
def ==(a: imag(64), b: imag(64)): bool
def ==(a: complex(64), b: complex(64)): bool
def ==(a: complex(128), b: complex(128)): bool
```

The result of `a == b` is true if `a` and `b` contain the same value; otherwise the result is false. The result of `a != b` is equivalent to `!(a == b)`.

The equality comparison operators are predefined over classes as follows:

```
def ==(a: object, b: object): bool
def !=(a: object, b: object): bool
```

The result of `a == b` is true if `a` and `b` reference the same storage location; otherwise the result is false. The result of `a != b` is equivalent to `!(a == b)`.

Default equality comparison operators are generated for records if the user does not define them. These operators are described in §16.4.

The equality comparison operators are predefined over strings as follows:

```
def ==(a: string, b: string): bool
def !=(a: string, b: string): bool
```

The result of `a == b` is true if the sequence of characters in `a` matches exactly the sequence of characters in `b`; otherwise the result is false. The result of `a != b` is equivalent to `!(a == b)`.

10.18 Miscellaneous Operators

This section describes several miscellaneous operators. These operators can be redefined over different types using operator overloading (§13.12).

10.18.1 The String Concatenation Operator

The string concatenation operator + is predefined over numeric, boolean, and enumerated types with strings. It casts its operands to string type and concatenates them together.

Example. The code

```
"result: "+i
```

where `i` is an integer appends the string representation of `i` to the string literal `"result: "`. If `i` is `3`, then the resulting string would be `"result: 3"`.

10.18.2 The By Operator

The operator `by` is predefined on ranges and arithmetic domains. It is described in §18.4.1 for ranges and §19.13.1 for domains.

10.18.3 The Range Count Operator

The operator `#` is predefined on ranges. It is described in §18.4.2.

10.19 Let Expressions

A let expression allows variables to be declared at the expression level and used within that expression. The syntax of a let expression is given by:

let-expression:
  let *variable-declaration-list* **in** *expression*

The scope of the variables is the let-expression.

Example. Let expressions are useful for defining variables in the context of expression. In the code

```
let x: real = a*b, y = x*x in 1/y
```

the value determined by `a*b` is computed and converted to type real if it is not already a real. The square of the real is then stored in `y` and the result of the expression is the reciprocal of that value.

10.20 Conditional Expressions

A conditional expression is given by the following syntax:

if-expression:
  if *expression* **then** *expression* **else** *expression*
  if *expression* **then** *expression*

The conditional expression is evaluated in two steps. First, the expression following the `if` keyword is evaluated. Then, if the expression evaluated to true, the expression following the `then` keyword is evaluated and taken to be the value of this expression. Otherwise, the expression following the `else` keyword is evaluated and taken to be the value of this expression. In both cases, the unselected expression is not evaluated.

The 'else' keyword can be omitted only when the conditional expression is immediately nested inside a forall expression. Such an expression is used to filter predicates as described in §10.21.1 and §25.2.3.

10.21 For Expressions

A for expression is given by the following syntax:

for-expression:
  for *index-var-declaration* **in** *iterator-expression* **do** *expression*
  for *iterator-expression* **do** *expression*

The for-expression evaluates a for-loop (§11.8) in the context of an expression and has the semantics of calling an iterator (§21) that yields the evaluated expressions on each iteration.

10.21.1 Filtering Predicates in For Expressions

A conditional expression that is immediately enclosed in a for expression does not require an else-part. Such a conditional expression filters the evaluated expressions and only returns an expression if the condition holds.

Example. The code

```
var A = for i in 1..10 do if i % 3 != 0 then i;
```

declares an array A that is initialized to the integers between 1 and 10 that are not divisible by 3.

11 Statements

Chapel is an imperative language with statements that may have side effects. Statements allow for the sequencing of program execution. They are as follows:

```
statement:
  block-statement
  expression-statement
  assignment-statement
  swap-statement
  conditional-statement
  select-statement
  while-do-statement
  do-while-statement
  for-statement
  label-statement
  break-statement
  continue-statement
  param-for-statement
  use-statement
  type-select-statement
  empty-statement
  return-statement
  yield-statement
  module-declaration-statement
  function-declaration-statement
  method-declaration-statement
  type-declaration-statement
  variable-declaration-statement
  remote-variable-declaration-statement
  on-statement
  cobegin-statement
  coforall-statement
  begin-statement
  sync-statement
  serial-statement
  atomic-statement
  forall-statement
```

Return statements are defined in §13.10. Yield statements are defined in §21.2. Module declaration statements are defined in §12. Function declaration statements are defined in §13. Method declaration statements are defined in §15.5. Type declaration statements are defined in §7. Variable declaration statements are defined in §8. Remote variable declaration statements are defined in §26.2.1. On statements are defined in §26.2. Cobegin, coforall, begin, sync, serial and atomic statements are defined in §24. Forall statements are defined in §25.

11.1 Blocks

A block is a statement or a possibly empty list of statements that form their own scope. A block is given by

```
block-statement:
  { statements_opt }
```

```
statements:
  statement
  statement statements
```

Variables defined within a block are local variables (§8.3).

The statements within a block are executed serially unless the block is in a cobegin statement (§24.2.1).

11.2 Expression Statements

The expression statement evaluates an expression solely for side effects. The syntax for an expression statement is given by

```
expression-statement:
  expression ;
```

11.3 Assignment Statements

An assignment statement assigns the value of an expression to another expression that can appear on the left-hand side of the operator, for example, a variable. Assignment statements are given by

```
assignment-statement:
  lvalue-expression assignment-operator expression

assignment-operator: one of
   = += -= *= /= %= **= &= |= ^= &&= ||= <<= >>=
```

The expression on the left-hand side of the assignment operator must be a valid lvalue (§10.10). It is evaluated before the expression on the right-hand side of the assignment operator, which can be any expression.

The assignment operators that contain a binary operator as a prefix is a short-hand for applying the binary operator to the left and right-hand side expressions and then assigning the value of that application to the already evaluated left-hand side. Thus, for example, `x += y` is equivalent to `x = x + y` where the expression `x` is evaluated once.

In a compound assignment, a cast to the type on the left-hand side is inserted before the simple assignment if the operator is a shift or both the right-hand side expression can be assigned to the left-hand side expression and the type of the left-hand side is a primitive type.

Rationale. This cast is necessary to handle `+=` where the type of the left-hand side is, for example, `int(8)` because the `+` operator is defined on `int(32)`, not `int(8)`.

Values of one primitive or enumerated type can be assigned to another primitive or enumerated type if an implicit coercion exists between those types (§9.1).

The validity and semantics of assigning between classes (§15.3), records (§16.1.6), unions (§17.3), tuples (§14.3), ranges (§18.3), domains (§19.8), and arrays (§20.5) is discussed in these later sections.

11.4 The Swap Statement

The swap statement indicates to swap the values in the expressions on either side of the swap operator. Since both expressions are assigned to, each must be a valid lvalue expression (§10.10).

swap-statement:
  lvalue-expression swap-operator lvalue-expression

swap-operator:
  <=>

To implement the swap operation, the compiler uses temporary variables as necessary.

Example. The following swap statement

```
var a, b: real;

a <=> b;
```

is semantically equivalent to:

```
const t = b;
b = a;
a = t;
```

11.5 The Conditional Statement

The conditional statement allows execution to choose between two statements based on the evaluation of an expression of `bool` type. The syntax for a conditional statement is given by

conditional-statement:
  if *expression* **then** *statement else-part*$_{opt}$
  if *expression block-statement else-part*$_{opt}$

else-part:
  else *statement*

A conditional statement evaluates an expression of bool type. If the expression evaluates to true, the first statement in the conditional statement is executed. If the expression evaluates to false and the optional else-clause exists, the statement following the `else` keyword is executed.

If the expression is a parameter, the conditional statement is folded by the compiler. If the expression evaluates to true, the first statement replaces the conditional statement. If the expression evaluates to false, the second statement, if it exists, replaces the conditional statement; if the second statement does not exist, the conditional statement is removed.

If the statement that immediately follows the optional `then` keyword is a conditional statement and it is not in a block, the else-clause is bound to the nearest preceding conditional statement without an else-clause.

Each statement embedded in the *conditional-statement* has its own scope whether or not an explicit block surrounds it.

11.6 The Select Statement

The select statement is a multi-way variant of the conditional statement. The syntax is given by:

> *select−statement*:
> **select** *expression* { *when−statements* }
>
> *when−statements*:
> *when−statement*
> *when−statement when−statements*
>
> *when−statement*:
> **when** *expression−list* **do** *statement*
> **when** *expression−list block−statement*
> **otherwise** *statement*
>
> *expression−list*:
> *expression*
> *expression* , *expression−list*

The expression that follows the keyword `select`, the select expression, is compared with the list of expressions following the keyword `when`, the case expressions, using the equality operator `==`. If the expressions cannot be compared with the equality operator, a compile-time error is generated. The first case expression that contains an expression where that comparison is `true` will be selected and control transferred to the associated statement. If the comparison is always `false`, the statement associated with the keyword `otherwise`, if it exists, will be selected and control transferred to it. There may be at most one `otherwise` statement and its location within the select statement does not matter.

Each statement embedded in the *when-statement* has its own scope whether or not an explicit block surrounds it.

11.7 The While and Do While Loops

There are two variants of the while loop in Chapel. The syntax of the while-do loop is given by:

> *while−do−statement*:
> **while** *expression* **do** *statement*
> **while** *expression block−statement*

The syntax of the do-while loop is given by:

> *do−while−statement*:
> **do** *statement* **while** *expression* ;

In both variants, the expression evaluates to a value of type `bool` which determines when the loop terminates and control continues with the statement following the loop.

The while-do loop is executed as follows:

1. The expression is evaluated.

2. If the expression evaluates to `false`, the statement is not executed and control continues to the statement following the loop.

3. If the expression evaluates to `true`, the statement is executed and control continues to step 1, evaluating the expression again.

The do-while loop is executed as follows:

1. The statement is executed.

2. The expression is evaluated.

3. If the expression evaluates to `false`, control continues to the statement following the loop.

4. If the expression evaluates to `true`, control continues to step 1 and the the statement is executed again.

In this second form of the loop, note that the statement is executed unconditionally the first time.

11.8 The For Loop

The for loop iterates over ranges, domains, arrays, iterators, or any class that implements an iterator named `these`. The syntax of the for loop is given by:

for-statement:
  for *index-var-declaration* **in** *iterator-expression* **do** *statement*
  for *index-var-declaration* **in** *iterator-expression block-statement*
  for *iterator-expression* **do** *statement*
  for *iterator-expression block-statement*

index-var-declaration:
  identifier
  tuple-grouped-identifier-list

iterator-expression:
  expression

The *index-var-declaration* declares new variables for the scope of the loop. It may specify a new identifier or may specify multiple identifiers grouped using a tuple notation in order to destructure the values returned by the iterator expression, as described in §14.4.3.

The *index-var-declaration* is optional and may be omitted if the indices do not need to be referenced in the loop.

If the iterator-expression is a tuple delimited by parentheses, the components of the tuple must support iteration, e.g., a tuple of arrays, and those components are iterated over using a zipper iteration defined in §11.8.1. If the iterator-expression is a tuple delimited by brackets, the components of the tuple must support iteration and these components are iterated over using a tensor product iteration defined in §11.8.2.

11.8.1 Zipper Iteration

When multiple iterators are iterated over in a zipper context, on each iteration, each expression is iterated over, the values are returned by the iterators in a tuple and assigned to the index, and the statement is executed.

The shape of each iterator, the rank and the extents in each dimension, must be identical.

Example. The output of

```
for (i, j) in (1..3, 4..6) do
  write(i, " ", j, " ");
```

is "1 4 2 5 3 6 ".

11.8.2 Tensor Product Iteration

When multiple iterators are iterated over in a tensor product context, they are iterated over as if they were nested in distinct for loops. There is no constraint on the iterators as there is in the zipper context.

Example. The output of

```
for (i, j) in [1..3, 4..6] do
  write(i, " ", j, " ");
```

is "1 4 1 5 1 6 2 4 2 5 2 6 3 4 3 5 3 6 ". The statement is equivalent to

```
for i in 1..3 do
  for j in 4..6 do
    write(i, " ", j, " ");
```

11.8.3 Parameter For Loops

Parameter for loops are unrolled by the compiler so that the index variable is a parameter rather than a variable. The syntax for a parameter for loop statement is given by:

param−iterator−expression:
  range−literal
  range−literal **by** *integer−literal*

param−for−statement:
  for param *identifier* **in** *param−iterator−expression* **do** *statement*
  for param *identifier* **in** *param−iterator−expression block−statement*

Parameter for loops are restricted to iteration over range literals with an optional by expression where the bounds and stride must be parameters. The loop is then unrolled for each iteration.

11.9 The Label, Break, and Continue Statements

The label-statement is used to name a specific loop which can then be the target of a break- or continue-statement. If a break- or continue-statement has no label, the target is the lexically inner-most loop. Labels can only be given to for-, while-do- and do-while-statements.

The syntax for label, break, and continue statements is given by:

label-statement:
  label *identifier statement*

break-statement:
  break *identifier*$_{opt}$;

continue-statement:
  continue *identifier*$_{opt}$;

If a break-statement is encountered, control will be transferred to after the associated loop. If a continue-statement is encountered, control will be transferred to the end of the associated loop, but still inside the loop. Break-statements cannot be used to break out of parallel loops. Neither break- nor continue-statements can cross out of cobegin-, coforall-, begin-, or sync-statements.

Example. In the following code, the index of the first element in each row of `A` that is equal to `findVal` is printed. Once a match is found, the continue statement is executed causing the outer loop to move to the next row.

```
label outer for i in 1..n {
  for j in 1..n {
    if A[i, j] == findVal {
      writeln("index: ", (i, j), " matches.");
      continue outer;
    }
  }
}
```

11.10 The Use Statement

The use statement makes symbols in modules available without accessing them via the module name. The syntax of the use statement is given by:

use-statement:
  use *module-name-list* ;

module-name-list:
  module-name
  module-name , *module-name-list*

module-name:
  identifier
  module-name . *module-name*

The use statement makes symbols in each listed module's scope available from the scope where the use statement occurs.

Symbols injected by a use statement are at an outer scope from those defined directly in the scope where the use statement occurs, but at a nearer scope than symbols defined in the scope containing the scope where the use statement occurs.

If used modules themselves use other modules, symbols are scoped according the depth of use statements followed to find them. It is an error for two variables, types, or modules to be defined at the same depth.

> *Open issue*. There is an expectation that this statement will be extended to allow the programmer to restrict which symbols are 'used' as well as to rename symbols that are used.

11.11 The Type Select Statement

A type select statement has two uses. It can be used to determine the type of a union, as discussed in §17.4. In its more general form, it can be used to determine the types of one or more values using the same mechanisms used to disambiguate function definitions. It syntax is given by:

type−select−statement:
  type select *expression−list* { *type−when−statements* }

type−when−statements:
  type−when−statement
  type−when−statement type−when−statements

type−when−statement:
  when *type−list* **do** *statement*
  when *type−list block−statement*
  otherwise *statement*

expression−list:
  expression
  expression , *expression−list*

type−list:
  type−specifier
  type−specifier , *type−list*

Call the expressions following the keyword `select`, the select expressions. The number of select expressions must be equal to the number of types following each of the `when` keywords. Like the select statement, one of the statements associated with a `when` will be executed. In this case, that statement is chosen by the function resolution mechanism. The select expressions are the actual arguments, the types following the `when` keywords are the types of the formal arguments for different anonymous functions. The function that would be selected by function resolution determines the statement that is executed. If none of the functions are chosen, the the statement associated with the keyword `otherwise`, if it exists, will be selected.

As with function resolution, this can result in an ambiguous situation. Unlike with function resolution, in the event of an ambiguity, the first statement in the list of when statements is chosen.

11.12 The Empty Statement

An empty statement has no effect. The syntax of an empty statement is given by

> *empty−statement*:
>   ;

12 Modules

Chapel supports modules to manage name spaces. A program consists of one or more modules. Every symbol, including variables, functions, and types, is associated with some module.

Module definitions are described in §12.1. The relation between files and modules is described in §12.2. Nested modules are described in §12.3. Module uses and explicit naming of symbols are described in §12.4. The execution of a program and module initialization is described in §12.5.

12.1 Module Definitions

A module is declared with the following syntax:

module−declaration−statement:
  module *module−identifier block−statement*

module−identifier:
  identifier

A module's name is specified after the `module` keyword. The *block−statement* opens the module's scope. Symbols defined in this block statement are defined in the module's scope and are called *top-level module symbols*.

Module declaration statements must be top-level statements within a module. A module that is declared within another module is called a nested module (§12.3).

12.2 Files and Implicit Modules

Multiple modules can be defined in the same file and need not bear any relation to the file in terms of their names.

Example. The following file contains two explicitly named modules (§12.4.1), MX and MY.

```
module MX {
  var x: int = 0;
  def printX() {
    writeln(x);
  }
}

module MY {
  var y: int = 0;
  def printY() {
    writeln(y);
  }
}
```

Module MX defines top-level module symbols x and printX, while MX defines top-level module symbols y and printY.

For any file that contains top-level statements other than module declarations, the file itself is treated as the module declaration. In this case, the module is implicit and takes its name from the file. If the file name is not a legal Chapel identifier, the module name cannot be used in a use statement.

Example. The following file, named myModule.chpl, defines an implicitly named module called myModule.

```
var x: int = 0;
var y: int = 0;

def printX() {
  writeln(x);
}
def printY() {
  writeln(y);
}
```

Module myModule defines the top-level module symbols x, y, printX, and printY.

12.3 Nested Modules

A nested module is a module that is defined within another module, the outer module. Nested modules automatically have access to all of the symbols in the outer module. However, the outer module needs to explicitly use a nested module to have access to its symbols.

A nested module can be used without using the outer module by explicitly naming the module in the use statement.

Example. The code

```
use libsci.blas;
```

uses a module named `blas` that is nested inside a module named `libsci`.

Files with both module declarations and top-level statements result in nested modules.

Example. The following file, named myModule.chpl, defines an implicitly named module called myModule, with nested modules MX and MY.

```
module MX {
  var x: int = 0;
}

module MY {
  var y: int = 0;
}

use MX, MY;

def printX() {
  writeln(x);
}

def printY() {
  writeln(y);
}
```

12.4 Using Modules

A module can be used by code outside of that module. This allows access to the top-level module symbols without the need for explicit naming (§12.4.1). Using modules is accomplished via the use statement as defined in §11.10.

12.4.1 Explicit Naming

All top-level module symbols can be named explicitly with the following syntax:

```
module-access-expression:
  module-identifier-list . identifier

module-identifier-list:
  module-identifier
  module-identifier . module-identifier-list
```

This allows two variables that have the same name to be distinguished based on the name of their module. Using explicit module naming in a function call restricts the set of candidate functions to those in the specified module.

If code refers to symbols that are defined by multiple modules, the compiler will issue an error. Explicit naming can be used to disambiguate the symbols in this case.

Example. In the following example,

```
module M1 {
  var x: int = 1;
  var y: int = -1;
  def printX() {
    writeln("M1's x is: ", x);
  }
  def printY() {
    writeln("M1's y is: ", y);
  }
}

module M2 {
  use M3;
  use M1;

  var x: int = 2;

  def printX() {
    writeln("M2's x is: ", x);
  }

  def main() {
    M1.x = 4;
    M1.printX();
    writeln(x);
    printX(); // This is not ambiguous
    printY(); // ERROR: This is ambiguous
  }
}
```

```
module M3 {
  var x: int = 3;
  var y: int = -3;
  def printY() {
    writeln("M3's y is: ", y);
  }
}
```

The call to printX() is not ambiguous because M2's definition shadows that of M1. On the other hand, the call to printY() is ambiguous because it is defined in both M1 and M3. This will result in a compiler error.

12.4.2 Module Initialization

Module initialization occurs at program start-up. All top-level statements in a module other than function and type declarations are executed during module initialization.

Example. In the code,

```
var x = foo();        // executed at module initialization
writeln("Hi!");       // executed at module initialization
def sayGoodbye {
  writeln("Bye!");    // not executed at module initialization
}
```

The function foo() will be invoked and its result assigned to x. Then “Hi!” will be printed.

Module initialization order is discussed in §12.5.2.

12.5 Program Execution

Chapel programs start by initializing all modules and then executing the main function (§12.5.1).

12.5.1 The *main* Function

The main function must be called `main` and must have zero arguments. It can be specified with or without parentheses. In any Chapel program, there is a single main function that defines the program's entry point. If a program defines multiple potential entry points, the implementation may provide a compiler flag that disambiguates between main functions in multiple modules.

Cray's Chapel Implementation. In the Cray Chapel compiler implementation, the *– –main-module* flag can be used to specify the module from which the main function definition will be used. Only modules passed to the compiler on the command line will be searched for main functions unless the *– –main-module* flag is used.

Example. If the following code is compiled without specifying a main module, it will yield an error.

```
module M1 {
  const x = 1;
  def main() {
    writeln("M", x, "'s main");
  }
}

module M2 {
  use M1;

  const x = 2;
  def main() {
    M1.main();
    writeln("M", x, "'s main");
  }
}
```

If M1 is specified as the main module, the program will output:

```
M1's main
```

If M2 is specified as the main module the program will output:

```
M1's main
M2's main
```

Notice that main is treated like just another function if it not in the main module and can be called as such.

To aid in exploratory programming, if the file(s) listed on the compiler's command line only define a single module, the module need not define a main function. In this case, a default main function is created to execute the module initialization code.

Example. The code

```
writeln("hello, world");
```

is a legal and complete Chapel program. Its initialization function, which also serves as the program's main function, is the top-level writeln() statement. The module declaration is taken to be the file as described in §12.2.

12.5.2 Module Initialization Order

Module initialization is performed using the following algorithm.

Module use can be represented by a directed graph over the modules. Starting from the module that defines the main function, modules are initialized in the order implied by a depth-first, post-order traversal of the graph. Modules are initialized in the order in which they appear in the program text. For nested modules, uses in the parent module are initialized before uses in the nested module.

Example. The code

```
module M1 {
  use M2.M3;
  use M2;
  writeln("In M1's initializer");
  def main() {
    writeln("In main");
  }
}

module M2 {
  use M4;
  writeln("In M2's initializer");
  module M3 {
    writeln("In M3's initializer");
  }
}

module M4 {
  writeln("In M4's initializer");
}
```

prints the following

```
In M4's initializer
In M2's initializer
In M3's initializer
In M1's initializer
In main
```

M1, the main module, uses M2.M3 and then M2, thus M2.M3 must be initialized. Because M2.M3 is a nested module, M4 (which is used by M2) must be initialized first. M2 itself is initialized, followed by M2.M3. Finally M1 is initialized, and the main function is run.

13 Functions

This section defines functions. Methods and iterators are functions and most of this section applies to them as well. They are defined separately in §21 and §15.5. Recursive and mutually-recursive functions are supported.

13.1 Function Calls

The syntax to call a function is given by:

```
call-expression:
  expression ( named-expression-list )
  expression [ named-expression-list ]
  parenthesesless-function-identifier

named-expression-list:
  named-expression
  named-expression , named-expression-list

named-expression:
  expression
  identifier = expression

parenthesesless-function-identifier:
  identifier
```

A *call-expression* is resolved to a particular function according to the algorithm for function resolution described in §13.14.

Functions can be called using either parentheses or brackets. The only difference in the call has to do with promotion and is discussed in §25.4.

Functions that are defined without parentheses are called without parentheses as defined by scope resolution. Functions without parentheses are discussed in §13.3.

A *named-expression* is an expression that may be optionally named. It provides an actual argument to the function being called. The optional *identifier* refers to a named formal argument described in §13.4.1.

13.2 Function Definitions

Functions are declared with the following syntax:

```
function-declaration-statement:
  def function-name argument-list_opt var-param-type-clause_opt where-clause_opt
    function-body

function-name:
  identifier
  operator-name
```

operator−name: *one of*
+ − ∗ / % ∗∗ ! == <= >= < > << >> & | ˆ ˜

argument−list:
(*formals*$_{opt}$)

formals:
formal
formal , *formals*

formal:
formal−intent$_{opt}$ *identifier formal−type*$_{opt}$ *default−expression*$_{opt}$
formal−intent$_{opt}$ *identifier formal−type*$_{opt}$ *variable−argument−expression*
formal−intent$_{opt}$ *tuple−grouped−identifier−list formal−type*$_{opt}$ *default−expression*$_{opt}$
formal−intent$_{opt}$ *tuple−grouped−identifier−list formal−type*$_{opt}$ *variable−argument−expression*

formal−type:
: *type−specifier*
: ? *identifier*$_{opt}$

default−expression:
= *expression*

variable−argument−expression:
... *expression*
... ? *identifier*$_{opt}$
...

formal−intent: *one of*
in out inout param type

var−param−type−clause:
var *return−type*$_{opt}$
const *return−type*$_{opt}$
param *return−type*$_{opt}$
type

return−type:
: *type−specifier*

where−clause:
where *expression*

function−body:
block−statement
return−statement

Functions do not require parentheses if they have no arguments. Such functions are described in §13.3.

Formal arguments can be grouped together using a tuple notation as described in §14.4.4.

Default expressions allow for the omission of actual arguments at the call site, resulting in the implicit passing of a default value. Default values are discussed in §13.4.2.

The intents `in`, `out`, and `inout` are discussed in §13.5. The intents `param` and `type` make a function generic and are discussed in §22.1. If the formal argument's type is omitted, generic, or prefixed with a question mark, the function is also generic and is discussed in §22.1.

Functions can take a variable number of arguments. Such functions are discussed in §13.6.

The optional *var−param−type−clause* defines a variable function, discussed in §13.7, or a parameter function, discussed in §13.8, or a type function, discussed in §13.9. By default, a function call cannot be treated as an lvalue and is constant. This may be explicitly specified via the keyword `const`.

Return types are optional and are discussed in §13.11.

The optional *where−clause* is only applicable if the function is generic. It is discussed in §22.4.

Function and operator overloading is supported in Chapel and is discussed in §13.12. Operator overloading is supported on the operators listed above (see *operator−name*).

13.3 Functions without Parentheses

Functions do not require parentheses if they have empty argument lists. Functions declared without parentheses around empty argument lists must be called without parentheses.

Example. Given the definitions

```
def foo { writeln("In foo"); }
def bar() { writeln("In bar"); }
```

the function `foo` can be called by writing `foo` and the function `bar` can be called by writing `bar()`. It is an error to apply parentheses to `foo` or omit them from `bar`.

13.4 Formal Arguments

Chapel supports an intuitive formal argument passing mechanism. An argument is passed by value unless it is a class, array, or domain, in which case it is passed by reference.

Intents (§13.5) may be used to override the default argument passing mechanism. This may result in assignments to and from the formal argument during a function call. For example, when passing an array by intent `in`, the actual argument array will be copied into the formal argument for use within the function.

13.4.1 Named Arguments

A formal argument can be named at the call site to explicitly map an actual argument to a formal argument.

Example. In the code

```
def foo(x: int, y: int) { writeln(x); writeln(y); }

foo(x=2, y=3);
foo(y=3, x=2);
```

named argument passing is used to map the actual arguments to the formal arguments. The two function calls are equivalent.

Named arguments are sometimes necessary to disambiguate calls or ignore arguments with default values. For a function that has many arguments, it is sometimes good practice to name the arguments at the call site for compiler-checked documentation.

13.4.2 Default Values

Default values can be specified for a formal argument by appending the assignment operator and a default expression to the declaration of the formal argument. If the actual argument is omitted from the function call, the default expression is evaluated when the function call is made and the evaluated result is passed to the formal argument as if it were passed from the call site.

Example. In the code

```
def foo(x: int = 5, y: int = 7) { writeln(x); writeln(y); }

foo();
foo(7);
foo(y=5);
```

default values are specified for the formal arguments `x` and `y`. The three calls to `foo` are equivalent to the following three calls where the actual arguments are explicit: `foo(5, 7)`, `foo(7, 7)`, and `foo(5, 5)`. The example `foo(y=5)` shows that we have to use a named argument for `y` in order to use the default value for `x` in the case when `x` appears earlier than `y` in the formal argument list.

13.5 Intents

Intents allow the actual arguments to be copied to a formal argument and also to be copied back.

13.5.1 The Blank Intent

If the intent is omitted, it is called a blank intent. In such a case, the value is copied in using the assignment operator. Thus classes are passed by reference and records are passed by value. Arrays and domains are an exception because assignment does not apply from the actual to the formal. Instead, arrays and domains are passed by reference.

With the exception of arrays, any argument that has blank intent cannot be assigned within the function.

13.5.2 The In Intent

If `in` is specified as the intent, the actual argument is copied to the formal argument as usual, but it may also be assigned to within the function. This assignment is not reflected back at the call site.

If an array is passed to a formal argument that has `in` intent, a copy of the array is made via assignment. Changes to the elements within the array are thus not reflected back at the call site.

13.5.3 The Out Intent

If `out` is specified as the intent, the actual argument is ignored when the call is made, but after the call, the formal argument is assigned to the actual argument at the call site. The actual argument must be a valid lvalue. The formal argument can be assigned to and read from within the function.

The formal argument cannot not be generic and is treated as a variable declaration.

13.5.4 The Inout Intent

If `inout` is specified as the intent, the actual argument is both passed to the formal argument as if the `in` intent applied and then copied back as if the `out` intent applied. The formal argument can be generic and takes its type from the actual argument. The formal argument can be assigned to and read from within the function.

13.6 Variable Length Argument Lists

Functions can be defined to take a variable number of arguments where those arguments can have any intent or can be types. A variable number of parameters is not supported. This allows the call site to pass a different number of actual arguments. There must be at least one actual argument.

If the variable argument expression contains an identifier prepended by a question mark, the number of actual arguments can vary, and the identifier will be bound to an integer parameter value indicating the number of arguments at a given call site. If the variable argument expression contains an expression without a question mark, that expression must evaluate to an integer parameter value requiring the call site to pass that number of arguments to the function.

Within the function, the formal argument that is marked with a variable argument expression is a tuple of the actual arguments.

Example. The code

```
def mywriteln(x ...?k) {
  for param i in 1..k do
    writeln(x(i));
}
```

defines a generic function called `mywriteln` that takes a variable number of arguments of any type and then writes them out on separate lines. The parameter for-loop (§11.8.3) is unrolled by the compiler so that `i` is a parameter, rather than a variable. This needs to be a parameter for-loop because the expression `x(i)` will have a different type on each iteration. The type of `x` can be specified in the formal argument list to ensure that the actuals all have the same type.

Example. Either or both the number of variable arguments and their types can be specified. For example, a basic function to sum the values of three integers can be wrtten as

```
def sum(x: int...3) return x(1) + x(2) + x(3);
```

Specifying the type is useful if it is important that each argument have the same type. Specifying the number is useful in, for example, defining a method on a class that is instantiated over a rank parameter.

Example. The function

```
def tuple(x ...) return x;
```

creates a generic function that returns tuples. When passed two or more actuals in a call, it is equivalent to building a tuple so the expressions `tuple(1, 2)` and `(1, 2)` are equivalent. When passed one actual, it builds a 1-tuple which is different than the evaluation of the parenthesized expression. Thus the expressions `tuple(1)` and `(1)` are not equivalent.

13.7 Variable Functions

A variable function is a function that can be assigned a value. Note that a variable function does not return a reference. That is, the reference cannot be captured.

A variable function is specified by following the argument list with the `var` keyword. A variable function must return an lvalue.

When a variable function is called on the left-hand side of an assignment statement or in the context of a call to a formal argument by out or inout intent, the lvalue that is returned by the function is assigned a value.

Variable functions support an implicit argument `setter` of type bool that is a compile-time constant (and can thus be folded). If the variable function is called in a context such that the returned lvalue is assigned a value, the argument `setter` is `true`; otherwise it is `false`. This argument is useful for controlling different behavior depending on the call site.

Example. The following code creates a function that can be interpreted as a simple two-element array where the elements are actually global variables:

```
var x, y = 0;

def A(i: int) var {
  if i < 0 || i > 1 then
    halt("array access out of bounds");
  if i == 0 then
    return x;
  else
    return y;
}
```

This function can be assigned to in order to write to the “elements” of the array as in

```
A(0) = 1;
A(1) = 2;
```

It can be called as an expression to access the “elements” as in

```
writeln(A(0) + A(1));
```

This code outputs the number `3`.

The implicit `setter` argument can be used to ensure, for example, that the second element in the pseudo-array is only assigned a value if the first argument is positive. To do this, add the following:

```
if setter && i == 1 && x <= 0 then
  halt("cannot assign value to A(1) if A(0) <= 0");
```

13.8 Parameter Functions

A parameter function is a function that returns a parameter expression. It is specified by following the function’s argument list by the keyword `param`. It is often, but not necessarily, generic.

It is a compile-time error if a parameter function does not return a parameter expression. The result of a parameter function is computed during compilation and the result is inlined at the call site.

Example. In the code

```
def sumOfSquares(param a: int, param b: int) param
  return a**2 + b**2;

var x: sumOfSquares(2, 3)*int;
```

the function `sumOfSquares` is a parameter function that takes two parameters as arguments. Calls to this function can be used in places where a parameter expression is required. In this example, the call is used in the declaration of a homogeneous tuple and so is required to be a parameter.

Parameter functions may not contain control flow that is not resolved at compile-time. This includes loops other than the parameter for loop §11.8.3 and conditionals with a conditional expressions that is not a parameter.

13.9 Type Functions

A type function is a function that returns a type. It is specified by following the function’s argument list by the keyword `type`. It is often, but not necessarily, generic.

It is a compile-time error if a type function does not return a type. The result of a type function is computed during compilation.

As with parameter functions, type functions may not contain control flow that is not resolved at compile-time. This includes loops other than the parameter for loop §11.8.3 and conditionals with a conditional expressions that is not a parameter.

Example. In the code

```
def myType(x) type {
  if numBits(x.type) <= 32 then return int;
  else return int(64);
}
```

the function `myType` is a type function that takes a single argument `x` and returns `int` if the number of bits used to represent `x` is less than or equal to 32, otherwise it returns `int(64)`. The function `numBits` is a param function defined in the Types module (§30.1.3).

13.10 The Return Statement

The return statement can only appear in a function. It exits that function, returning control to the point at which that function was called. It can optionally return a value. The syntax of the return statement is given by

return-statement:
  return *expression*$_{opt}$;

Example. The following code defines a function that returns the sum of three integers:

```
def sum(i1: int, i2: int, i3: int)
  return i1 + i2 + i3;
```

13.11 Return Types

A function can optionally return a value. If the function does not return a value, its return type is `void`. Specifying the return type in a function declaration is optional.

13.11.1 Explicit Return Types

If a return type is specified and is not `void`, the values that the function returns via return statements must be assignable to a variable of the return type. For variable functions (§13.7), the return type must match the type returned in all of the return statements exactly, when checked after generic instantiation and parameter folding (if applicable).

13.11.2 Implicit Return Types

If a return type is not specified, it will be inferred from the return statements. Given the types that are returned by the different statements, if exactly one of those types can be a target, via implicit conversions, of every other type, then that is the inferred return type. Otherwise, it is an error. For variable functions (§13.7), every return statement must return the same exact type and it becomes the inferred type. For functions without any return statements, the return type is `void`.

13.12 Function Overloading

Functions that have the same name but different argument lists are called overloaded functions. Function calls to overloaded functions are resolved according to the algorithm in §13.14.

Operator overloading is achieved by defining a function with a name specified by that operator. The operators that may be overloaded are listed in the following table:

arity	**operators**
unary	`+ - ! ~`
binary	`+ - * / % ** ! == <= >= < > << >> & | ^` by

The arity and precedence of the operator must be maintained when it is overloaded. Operator resolution follows the same algorithm as function resolution.

13.13 Nested Functions

A function defined in another function is called a nested function. Nesting of functions may be done to arbitrary degrees, i.e., a function can be nested in a nested function.

Nested functions are only visible to function calls within the lexical scope in which they are defined.

Nested functions may refer to variables defined in the function(s) in which they are nested.

13.14 Function Resolution

Given a function call, the function that the call resolves to is determined according to the following algorithm:

- Identify the set of visible functions for the function call. A *visible function* is any function that satisfies the criteria in §13.14.1. If no visible function can be found, the compiler will issue an error stating that the call cannot be resolved.
- From the set of visible functions for the function call, determine the set of candidate functions for the function call. A *candidate function* is any function that satisfies the criteria in §13.14.2. If no candidate function can be found, the compiler will issue an error stating that the call cannot be resolved. If exactly one candidate function is found, this is determined to be the function.
- From the set of candidate functions, the most specific function is determined. The most specific function is a candidate function that is *more specific* than every other candidate function as defined in §13.14.3. If there is no function that is more specific than every other candidate function, the compiler will issue an error stating that the call is ambiguous.

13.14.1 Determining Visible Functions

Given a function call, a function is determined to be a *visible function* if the name of the function is the same as the name of the function call and the function is defined in the same scope as the function call or a lexical outer scope of the function call, or if the function is defined in a module that is used from the same scope as the function call or a lexical outer scope fo the function call. Function visibility in generic functions is discussed in §22.2.

13.14.2 Determining Candidate Functions

Given a function call, a function is determined to be a *candidate function* if there is a *valid mapping* from the function call to the function and each actual argument is mapped to a formal argument that is a *legal argument mapping*.

Valid Mapping The following algorithm determines a valid mapping from a function call to a function if one exists:

- Each actual argument that is passed by name is matched to the formal argument with that name. If there is no formal argument with that name, there is no valid mapping.
- The remaining actual arguments are mapped in order to the remaining formal arguments in order. If there are more actual arguments then formal arguments, there is no valid mapping. If any formal argument that is not mapped to by an actual argument does not have a default value, there is no valid mapping.
- The valid mapping is the mapping of actual arguments to formal arguments plus default values to formal arguments that are not mapped to by actual arguments.

Legal Argument Mapping An actual argument of type T_A can be mapped to a formal argument of type T_F if any of the following conditions hold:

- T_A and T_F are the same type.
- There is an implicit coercion from T_A to T_F.
- T_A is derived from T_F.
- T_A is scalar promotable to T_F.

13.14.3 Determining More Specific Functions

Given two functions F_1 and F_2, the more specific function is determined by the following steps:

- If F_1 does not require promotion and F_2 does require promotion, then F_1 is more specific.
- If F_2 does not require promotion and F_1 does require promotion, then F_2 is more specific.

- If at least one of the legal argument mappings to F_1 is a *more specific argument mapping* than the corresponding legal argument mapping to F_2 and none of the legal argument mappings to F_2 is a more specific argument mapping than the corresponding legal argument mapping to F_1, then F_1 is more specific.
- If at least one of the legal argument mappings to F_2 is a *more specific argument mapping* than the corresponding legal argument mapping to F_1 and none of the legal argument mappings to F_1 is a more specific argument mapping than the corresponding legal argument mapping to F_2, then F_2 is more specific.
- If F_1 shadows F_2, then F_1 is more specific.
- If F_2 shadows F_1, then F_2 is more specific.
- If F_1 has a where clause and F_2 does not have a where clause, then F_1 is more specific.
- If F_2 has a where clause and F_1 does not have a where clause, then F_2 is more specific.
- Otherwise neither function is more specific.

Given an argument mapping, M_1, from an actual argument, A, of type T_A to a formal argument, $F1$, of type T_{F1} and an argument mapping, M_2, from the same actual argument to a formal argument, $F2$, of type T_{F2}, the more specific argument mapping is determined by the following steps:

- If T_{F1} and T_{F2} are the same type, $F1$ is an instantiated parameter, and $F2$ is not an instantiated parameter, M_1 is more specific.
- If T_{F1} and T_{F2} are the same type, $F2$ is an instantiated parameter, and $F1$ is not an instantiated parameter, M_2 is more specific.
- If M_1 does not require scalar promotion and M_2 requires scalar promotion, M_1 is more specific.
- If M_1 requires scalar promotion and M_2 does not require scalar promotion, M_2 is more specific.
- If T_{F1} and T_{F2} are the same type, $F1$ is generic, and $F2$ is not generic, M_1 is more specific.
- If T_{F1} and T_{F2} are the same type, $F2$ is generic, and $F1$ is not generic, M_2 is more specific.
- If $F1$ is not generic over all types and $F2$ is generic over all types, M_1 is more specific.
- If $F1$ is generic over all types and $F2$ is not generic over all types, M_2 is more specific.
- If T_A and T_{F1} are the same type and T_A and T_{F2} are not the same type, M_1 is more specific.
- If T_A and T_{F1} are not the same type and T_A and T_{F2} are the same type, M_2 is more specific.
- If T_{F1} is derived from T_{F2}, then M_1 is more specific.
- If T_{F2} is derived from T_{F1}, then M_2 is more specific.
- If there is an implicit coercion from T_{F1} to T_{F2}, then M_1 is more specific.
- If there is an implicit coercion from T_{F2} to T_{F1}, then M_2 is more specific.
- If T_{F1} is any `int` type and T_{F2} is any `uint` type, M_1 is more specific.
- If T_{F2} is any `int` type and T_{F1} is any `uint` type, M_2 is more specific.
- Otherwise neither mapping is more specific.

14 Tuples

A tuple is an ordered set of components that allows for the specification of a light-weight collection of values. As the examples in this chapter illustrate, tuples are a boon to the Chapel programmer. In addition to making it easy to return multiple values from a function, tuples help to support multidimensional indices, to group arguments to functions, and to specify mathematical concepts.

14.1 Tuple Types

A tuple type is defined by a fixed number (a compile-time constant) of component types. It can be specified by a parenthesized, comma-separated list of types. The number of types in the list defines the size of the tuple; the types themselves specify the component types.

The syntax of a tuple type is given by:

```
tuple-type:
  ( type-specifier , type-list )

type-list:
  type-specifier
  type-specifier , type-list
```

A homogeneous tuple is a special-case of a general tuple where the types of the components are identical. Homogeneous tuples have fewer restrictions for how they can be indexed (§14.5). Homogeneous tuple types can be defined using the above syntax, or they can be defined as a product of an integral parameter (a compile-time constant integer) and a type. This latter specification is implemented by overloading `*` with the following prototype:

```
def *(param p: int, type t) type
```

Rationale. Homogeneous tuples require the size to be specified as a parameter (compile-time constant). This avoids any overhead associated with storing the runtime size in the tuple. It also avoids the question as to whether a non-parameter size should be part of the type of the tuple. If a programmer requires a non-parameter value to define a data structure, an array may be a better choice.

Example. The statement

```
var x1: (string, real),
    x2: (int, int, int),
    x3: 3*int;
```

defines three variables. Variable `x1` is a 2-tuple with component types `string` and `real`. Variables `x2` and `x3` are homogeneous 3-tuples with component type `int`. The types of `x2` and `x3` are identical even though they are specified in different ways.

Note that if a single type is delimited by parentheses, the parentheses only impact precedence. Thus `(int)` is equivalent to `int`. Nevertheless, tuple types with a single component type are legal and useful. One way to specify a 1-tuple is to use the overloaded `*` operator since every 1-tuple is trivially a homogeneous tuple.

Rationale. Like parentheses around expressions, parentheses around types are necessary for grouping in order to avoid the default precedence of the grammar. Thus it is not the case that we would always want to create a tuple. The type `3*(3*int)` specifies a 3-tuple of 3-tuples of integers rather than a 3-tuple of 1-tuples of 3-tuples of integers. The type `3*3*int`, on the other hand, specifies a 9-tuple of integers.

14.2 Tuple Values

A value of a tuple type attaches a value to each component type. Tuple values can be specified by a parenthesized, comma-separated list of expressions. The number of expressions in the list defines the size of the tuple; the types of these expressions specify the component types of the tuple.

The syntax of a tuple expression is given by:

```
tuple-expression:
  ( expression , expression-list )

expression-list:
  expression
  expression , expression-list
```

Example. The statement

```
var x1: (string, real) = ("hello", 3.14),
    x2: (int, int, int) = (1, 2, 3),
    x3: 3*int = (4, 5, 6);
```

defines three variables. Variable `x1` is a 2-tuple with component types `string` and `real`. It is initialized such that the first component is `"hello"` and the second component is `3.14`. Variables `x2` and `x3` are homogeneous 3-tuples with component type `int`. Their initialization expressions specify 3-tuples of integers.

The function

```
def tuple(x...) return x;
```

is defined in the standard context to create arbitrary tuples.

Example. The statement

```
var x1 =      ("hello", 3.14),
    x2 = tuple("hello", 3.14),
    x3 = tuple(1);
```

creates two identical tuples `x1` and `x2`, and a 1-tuple of an integer `x3` with its component initialized to `1`.

Note that if a single expression is delimited by parentheses, the parentheses only impact precedence. Thus `(1)` is equivalent to `1`. Tuple expressions with a single component are legal and useful. As seen in the example above, one way to specify a 1-tuple is to use the standard `tuple` function.

Tuple expressions are evaluated similarly to function calls where the arguments are all generic with no explicit intent. So a tuple expression containing an array does not copy the array. The semantics regarding passing tuples to functions is forthcoming.

14.3 Tuple Assignment

In tuple assignment, the components of the tuple on the left-hand side of the assignment operator are each assigned the components of the tuple on the right-hand side of the assignment. These assignments occur in component order (component one followed by component two, etc.).

14.4 Tuple Destructuring

Tuples can be split into their components in four ways:

- In assignment where multiple expression on the left-hand side of the assignment operator are grouped using tuple notation.
- In variable declarations where multiple variables in a declaration are grouped using tuple notation.
- In for, forall, and coforall loops (statements and expressions) where multiple indices in a loop are grouped using tuple notation.
- In function calls where multiple formal arguments in a function declaration are grouped using tuple notation.
- In an expression context that accepts a comma-separated list of expressions where a tuple expression is expanded in place using the tuple expansion expression.

14.4.1 Splitting a Tuple with Assignment

When multiple expression on the left-hand side of an assignment operator are grouped using tuple notation, the tuple on the right-hand side is split into its components. The number of grouped expressions must be equal to the size of the tuple on the right-hand side. In addition to the usual assignment evaluation order of left to right, the assignment is evaluated in component order.

Example. The code

```
var a, b, c: int;
(a, (b, c)) = (1, (2, 3));
```

defines three integer variables `a`, `b`, and `c`. The second line then splits the tuple `(1, (2, 3))` such that `1` is assigned to `a`, `2` is assigned to `b`, and `3` is assigned to `c`.

Example. The code

```
var A = [i in 1..4] i;
writeln(A);
(A(1..2), A(3..4)) = (A(3..4), A(1..2));
writeln(A);
```

creates a non-distributed, one-dimensional array containing the four integers from `1` to `4`. Line 2 outputs `1 2 3 4`. Line 3 does what appears to be a swap of array slices. However, because the tuple is created with array aliases (like a function call), the assignment to the second component uses the values just overwritten in the assignment to the first component. Line 4 outputs `3 4 3 4`.

When splitting a tuple with assignment, the expressions that are grouped using the tuple notation may be omitted. In this case, the expression on the right-hand side of the assignment operator is evaluated, but its value is not assigned.

Example. The code

```
def f()
  return (1, 2);

var x: int;
(x, ) = f();
```

defines a function that returns a 2-tuple, declares an integer variable `x`, calls the function, assigns the first component in the returned tuple to `x`, and ignores the second component in the returned tuple. The value of `x` becomes `1`.

14.4.2 Splitting a Tuple into Multiple Variables

When multiple variables in a declaration are grouped using tuple notation, the tuple type and/or tuple initialization expression are split into their components. The number of grouped variables must be equal to the size of the tuple type and/or tuple initialization expression. The variables are initialized in component order.

The syntax of grouped variable declarations is defined in §8.1.

Example. The code

```
var (a, (b, c)) = (1, (2, 3));
```

defines three integer variables `a`, `b`, and `c`. It splits the tuple `(1, (2, 3))` such that `1` initializes `a`, `2` initializes `b`, and `3` initializes `c`.

Grouping variable declarations using tuple notation allows a 1-tuple to be destructured by enclosing a single variable declaration in parentheses.

Example. The code

```
var (a) = tuple(1);
```

initialize the new variable `a` to 1.

When splitting a tuple into multiple variable declarations, the variables that are grouped using the tuple notation may be omitted. In this case, a variable is not defined for any omitted component.

Example. The code

```
def f()
  return (1, 2);

var (x, ) = f();
```

defines a function that returns a 2-tuple, calls the function, declares and initializes variable `x` to the first component in the returned tuple, and ignores the second component in the returned tuple. The value of `x` is initialized to `1`.

14.4.3 Splitting a Tuple into Mutiple Indices

When multiple indices in a loop are grouped using tuple notation, the values returned by the iterator are split into their components. The number of grouped indices must be equal to the size of the tuple returned by the iterator.

Example. The code

```
def bar() {
  yield (1, 1);
  yield (2, 2);
}

for (i,j) in bar() do
  writeln(i+j);
```

defines a simple iterator that yields two 2-tuples before completing. The for-loop uses a tuple notation to group two indices that take their values from the iterator.

When splitting a tuple into multiple indices, the indices that are grouped using the tuple notation may be omitted. In this case, a new index is not defined for any omitted component. The iterator is evaluated as if an index were defined.

14.4.4 Splitting a Tuple into Multiple Formal Arguments

When multiple formal arguments in a function declaration are grouped using tuple notation, the actual expression is split into its components during a function call. The number of grouped formal arguments must be equal to the size of the actual tuple expression. The actual arguments are passed in component order to the formal arguments.

The syntax of grouped formal arguments is defined in §13.2.

Example. The function

```
def f(x: int, (y, z): (int, int)) {
  // body
}
```

is defined to take an integer value and a 2-tuple of integer values. The 2-tuple is split when the function is called into two formals. A call may look like the following:

```
f(1, (2, 3));
```

An implicit where clause is created when arguments are grouped using a tuple notation to ensure that the function is called with an actual tuple of the correct size. Grouping arguments in tuples may be arbitrarily nested. Functions with tuple-grouped arguments may not be called using named-argument passing on the tuple-grouped arguments. In addition, tuple-grouped arguments may not be specified individually with types or default values (only in aggregate). They may not be specified with any qualifier appearing before the group of arguments (or individual arguments) such as `inout` or `type`. They may not be followed by `...` to indicate that there are a variable number of them.

Example. The function `f` defined as

```
def f((x, (y, z))) {
  writeln((x, y, z));
}
```

is equivalent to the function `g` defined as

```
def g(t) where isTuple(t) && t.size == 2 && isTuple(t(2)) && t(2).size == 2 {
  writeln((t(1), t(2)(1), t(2)(2)));
}
```

except without the definition of the argument name `t`.

Grouping formal arguments using tuple notation allows a 1-tuple to be destructured by enclosing a single formal argument in parentheses.

Example. The empty function

```
def f((x)) { }
```

accepts a 1-tuple actual with any component type.

When splitting a tuple into multiple formal arguments, the arguments that are grouped using the tuple notation may be omitted. In this case, a new argument is not defined for any omitted component. The call is evaluated as if an argument were defined.

14.4.5 Splitting a Tuple via Tuple Expansion

Tuples can be expanded in place using the following syntax:

tuple-expand-expression:
  (... *expression*)

In this expression, the tuple defined by *expression* is expanded in place to represent its components. This can only be used in a context where a comma-separated list of components is valid.

Example. Given two 2-tuples

```
var x1 = (1, 2.0), x2 = ("three", "four");
```

the following statement

```
var x3 = ((...x1), (...x2));
```

creates the 4-tuple `x3` with the value `(1, 2.0, "three", "four")`.

Example. The following code defines two functions, a function `first` that returns the first component of a tuple and a function `rest` that returns a tuple containing all of the components of a tuple except for the first:

```
def first(t) where isTuple(t) {
  return t(1);
}
def rest(t) where isTuple(t) {
  def helper(first, rest...)
    return rest;
  return helper((...t));
}
```

14.5 Tuple Indexing

A tuple may be accessed by an integral parameter (compile-time constant) as if it were an array. Tuples are *1-based* so the first component in the tuple is accessed by the value `1`, and so forth.

Example. The loop

```
var tuple = (1, 2.0, "three");
for param i in 1..3 do
  writeln(tuple(i));
```

uses a param loop to output the components of a tuple.

Homogeneous tuples may be accessed by integral values that are not necessarily compile-time constants.

Example. The loop

```
var tuple = (1, 2, 3);
for i in 1..3 do
  writeln(tuple(i));
```

uses a serial loop to output the components of a homogeneous tuple. Since the index is not a compile-time constant, this would result in an error were tuple not homogeneous.

Rationale. Non-homogeneous tuples can only be accessed by compile-time constants so that the type of the expression is statically known.

14.6 Tuple Operators

14.6.1 Unary Operators

The unary operators `+`, `-`, `~`, and `!` are overloaded on tuples by applying the operator to each argument component and returning the results as a new tuple.

The size of the result tuple is the same as the size of the argument tuple. The type of each result component is the result type of the operator when applied to the corresponding argument component.

14.6.2 Binary Operators

The binary operators `+`, `-`, `*`, `/`, `%`, `**`, `&`, `|`, `^`, `<<`, and `>>` are overloaded on tuples by applying them to pairs of the respective argument components and returning the results as a new tuple. The sizes of the two argument tuples must be the same.

The size of the result tuple is the same as the argument tuples. The type of each result component is the result type of the operator when applied to the corresponding pair of the argument components.

Example. The code

```
var x = (1, 1, 1) + (2, 2.0, "2");
```

creates a 3-tuple of an int, a real and a string with the value `(3, 3.0, "12")`.

14.6.3 Relational Operators

The relational operators `>`, `>=`, `<`, `<=`, `==`, and `!=` are defined over tuples of matching size. They return a single boolean value indicating whether the two arguments satisfy the corresponding relation.

The operators `>`, `>=`, `<`, and `<=` check the corresponding lexicographical order based on pair-wise comparisons between the argument tuples' components. The operators `==` and `!=` check whether the two arguments are pair-wise equal or not. The relational operators on tuples may be short-circuiting, i.e. they may execute only the pair-wise comparisons that are necessary to determine the result.

Example. The code

```
var x = (1, 1, 0) > (1, 0, 1);
```

creates a variable initialized to `true`. After comparing the first components and determining they are equal, the second components are compared to determine that the first tuple is greater than the second tuple.

14.7 Predefined Functions and Methods on Tuples

def *Tuple*`.size` **param**

Returns the size of the tuple.

def `isHomogeneousTuple(t:` *Tuple*`)` **param**

Returns true if `t` is a homogeneous tuple; otherwise false.

def `isTuple(t:` *Tuple*`)` **param**

Returns true if `t` is a tuple; otherwise false.

def `isTupleType(`**type** `t)` **param**

Returns true if `t` is a tuple of types; otherwise false.

def `max(`**type** `t)` **where** `isTupleType(t)`

Returns a tuple of type `t` with each component set to the maximum value that can be stored in its position.

def `min(`**type** `t)` **where** `isTupleType(t)`

Returns a tuple of type `t` with each component set to the minimum value that can be stored in its position.

15 Classes

Classes are data structures with associated state and methods. Storage for a class instance, or object, is allocated independently of the scope of the variable that refers to it. An object is created by calling a class constructor (§15.6), which allocates storage, initializes it, and returns a reference to the newly-created object. Storage can be reclaimed by deleting the object (§15.11).

A class declaration (§15.2) generates a class type (§15.1). A variable of a class type can refer to an instance of that class or any of its derived classes.

A class is generic if it has generic fields. Generic classes and fields are discussed in §22.3.

15.1 Class Types

The syntax of a class type is summarized as follows:

class−type:
  identifier
  identifier (*named−expression−list*)

For non-generic classes, the class name is sufficient to specify the type. Generic classes must be instantiated to serve as a fully-specified type, for example to declare a variable. This is done with type constructors, which are defined in Section 22.3.4.

15.2 Class Declarations

A class is defined with the following syntax:

class−declaration−statement:
  class *identifier class−inherit−list*$_{opt}$ {
    class−statement−list$_{opt}$ }

class−inherit−list:
  : *class−type−list*

class−type−list:
  class−type
  class−type , *class−type−list*

class−statement−list:
  class−statement
  class−statement class−statement−list

class−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

A *class-declaration-statement* defines a new type symbol specified by the identifier. Classes inherit data and functionality from other classes if the *inherit-type-list* is specified. Inheritance is described in §15.8.

The body of a class declaration consists of a sequence of statements where each of the statements either defines a variable (called a field), a function (called a method), or a type alias. In addition, empty statements are allowed in class declarations, and they have no effect.

If a class contains a type alias or a parameter field, or it contains a variable or constant without a specified type or an initialization expression, the class is generic. Generic classes are described in §22.3.

15.3 Class Assignment

Classes are assigned by reference. After an assignment from one variable of class type to another, the variables reference the same class instance, i.e. the same storage location.

15.4 Class Fields

Variable declarations within a class define fields within that class. Parameter fields make a class generic. Variable and constant fields define the storage associated with a class.

Example. The code

```
class Actor {
  var name: string;
  var age: uint;
}
```

defines a new class type called `Actor` that has two fields: the string field `name` and the unsigned integer field `age`.

15.4.1 Class Field Accesses

The field in a class is accessed via a member access expression as described in §10.7. Fields in a class can be modified via an assignment statement where the left-hand side of the assignment is a member access expression. Accessing a parameter field returns a parameter.

Example. Given a variable `anActor` of type `Actor`, defined above, the code

```
var s: string = anActor.name;
anActor.age = 27;
```

reads the field `name` and assigns the value to the variable `s`, and assigns the storage location in the object `anActor` associated with the field `age` the value `27`.

15.5 Class Methods

A method is a function or iterator that is bound to a class. A method is called by passing an instance of the class to the method via a special syntax that is similar to a field access.

15.5.1 Class Method Declarations

Methods are declared with the following syntax:

> *method-declaration-statement*:
> **def** *param-clause*$_{opt}$ *type-binding function-name argument-list*$_{opt}$ *var-param-type-clause*$_{opt}$
> *return-type*$_{opt}$ *where-clause*$_{opt}$ *function-body*
>
> *param-clause*:
> **param**
>
> *type-binding*:
> *identifier* .

If a method is declared within the lexical scope of a class, record, or union, the type binding can be omitted and is taken to be the innermost class, record, or union that the method is defined in. If a method declaration contains the optional *param-clause*, it implies that it can only be applied to param objects of the given type binding.

15.5.2 Class Method Calls

A method is called by using the member access syntax as described in §10.7 where the accessed expression is the name of the method.

Example. A method to output information about an instance of the `Actor` class can be defined as follows:

```
def Actor.print() {
  writeln("Actor ", name, " is ", age, " years old");
}
```

This method can be called on an instance of the `Actor` class, `anActor`, by writing `anActor.print()`.

15.5.3 The *this* Reference

The instance of a class is passed to a method using special syntax. It does not appear in the argument list to the method. The reference `this` is an alias to the instance of the class on which the method is called.

Example. Let class `C`, method `foo`, and function `bar` be defined as

```
class C {
  def foo() {
    bar(this);
  }
}
def bar(c: C) { writeln(c); }
```

Then given an instance of `C` called `c`, the method call `c.foo()` results in a call to `bar` where the argument is `c`.

15.5.4 The *this* Method

A method declared with the name `this` allows a class to be "indexed" similarly to how an array is indexed. Indexing into a class has the semantics of calling a method on the class named `this`. There is no other way to call a method called `this`. The `this` method must be declared with parentheses even if the argument list is empty.

> *Example*. In the following code, the `this` method is used to create a class that acts like a simple array that contains three integers indexed by 1, 2, and 3.

```
class ThreeArray {
  var x1, x2, x3: int;
  def this(i: int) var {
    select i {
      when 1 do return x1;
      when 2 do return x2;
      when 3 do return x3;
    }
    halt("ThreeArray index out of bounds: ", i);
  }
}
```

15.5.5 The *these* Method

A method declared with the name `these` allows a class to be "iterated over" similarly to how a domain or array is iterated over. Using a class in the context of a loop where an *iterator-expression* is expected has the semantics of calling a method on the class named `these`.

> *Example*. In the following code, the `these` method is used to create a class that acts like a simple array that can be iterated over and contains three integers.

```
class ThreeArray {
  var x1, x2, x3: int;
  def these() var {
    yield x1;
    yield x2;
    yield x3;
  }
}
```

15.6 Class Constructors

Class instances are created by invoking class constructors. A class constructor is a method with the same name as the class. It is invoked by the `new` operator, where the class name and constructor arguments are preceded with the `new` keyword.

When the constructor is called, memory is allocated to store a class instance, its fields are initialized to default values (field defaults when specified in the class declaration, type defaults otherwise), then the constructor method is invoked on this newly-created instance.

If the program declares a class constructor method, it is a user-defined constructor. There is also the default constructor that is automatically created for each class. The default constructor is invoked by the `new` operator when there are no user-defined constructors for that class, otherwise it cannot be accessed explicitly.

15.6.1 User-Defined Constructors

A user-defined constructor is a constructor method explicitly declared in the program. The usual function resolution mechanism (§13.14) is applied to determine which user-defined constructor to invoke.

Any user-defined constructor begins with an implicit call to the default constructor (§15.6.2) for that class. This call passes no arguments explicitly, so each field is set to the initializer expression if given in the field's declaration and to the default value for the field's type otherwise.

Example. The following example shows a class with two constructors:

```
class MessagePoint {
  var x, y: real;
  var message: string;

  def MessagePoint(x: real, y: real) {
    this.x = x;
    this.y = y;
    this.message = "a point";
  }

  def MessagePoint(message: string) {
    this.x = 0;
    this.y = 0;
    this.message = message;
  }
}  // class MessagePoint

// create two objects
var mp1 = new MessagePoint(1.0,2.0);
var mp2 = new MessagePoint("point mp2");
```

The first constructor lets the user specify the initial coordinates and the second constructor lets the user specify the initial message when creating a MessagePoint.

Constructors for generic classes (§22.3) handle certain arguments differently and may need to satisfy additional requirements. See Section 22.3.7 for details.

15.6.2 The Default Constructor

The default constructor is automatically created for every class in the Chapel program. It has one argument for every field in the class with the argument name matching the field's name. This includes fields inherited from superclasses, type aliases and parameter fields, if any. The order of the arguments matches the order of the field declarations within the class, with the arguments for a superclass's fields occurring before the arguments for a subclass's fields. Each argument has a default value. For a field that has an initializer expression, the default value is that expression. Otherwise, it is the default value for the field's type (§8.1.1). This rule does not apply to generic fields without initializers, which are discussed in Section 22.3.6.

The default constructor for a class can be invoked by a `new` operator only when the class does not have any user-defined constructors. In this case the usual function resolution mechanism (§13.14) determines whether an invocation of the default constructor is legal.

When invoked, the default constructor initializes each field in the class to the value of the corresponding actual. This ensures that upon return from the constructor it is safe to use the object being created, as each field contains a legal value for its type. The initialization is done in the order the fields are declared, with a superclass's fields initialized before a subclass's fields.

The program may define a constructor with the same arguments and types as the default constructor for that class. If there is such a constructor, it is a user-defined constructor and is distinct from the default constructor.

Example. Given the class

```
class C {
  var x: int;
  var y: real = 3.14;
  var z: string = "Hello, World!";
}
```

there are no user-defined constructors for `C`, so `new` operators will invoke `C`'s default constructor. The `x` argument of the default constructor has the default value `0`. The `y` and `z` arguments have the default values `3.14` and `"Hello, World`"!, respectively.

`C` instances can be created by calling the default constructor as follows:

- The call `new C()` is equivalent to `C(0,3.14,"Hello, World`")!.
- The call `new C(2)` is equivalent to `C(2,3.14,"Hello, World`")!.
- The call `new C(z="")` is equivalent to `C(0,3.14,"")`.
- The call `new C(2, z="")` is equivalent to `C(2,3.14,"")`.
- The call `new C(0,0.0,"")` specifies the initial values for all fields explicitly.

15.7 Variable Getter Methods

All field accesses are resolved via getters that are variable methods (§13.7) defined in the class with the same name as the field. The default getter is defined to simply return the field if the user does not define their own.

Example. In the code

```
class C {
  var setCount: int;
  var x: int;
  def x var {
    if setter then
      setCount += 1;
    return x;
  }
}
```

an explicit variable getter method is defined for field `x`. It returns the field `x` and increments another field that records the number of times x was assigned a value.

15.8 Inheritance

A “derived” class can inherit from one or more other classes by specifying those classes, the base classes, following the name of the derived class in the declaration of the derived class. When inheriting from multiple base classes, only one of the base classes may contain fields. The other classes can only define methods. Note that a class can still be derived from a class that contains fields which is itself derived from a class that contains fields.

15.8.1 The object Class

All classes are derived from the `object` class either directly, or through the classes they are derived from. A variable of type `object` can hold a reference to an object of any class type.

15.8.2 Accessing Base Class Fields

A derived class contains data associated with the fields in its base classes. The fields can be accessed in the same way that they are accessed in their base class unless the getter or setter method is overridden in the derived class, as discussed in §15.8.5.

15.8.3 Derived Class Constructors

Derived class constructors automatically call the default constructor of the base class. There is an expectation that a more standard way of chaining constructor calls will be supported.

15.8.4 Shadowing Base Class Fields

A field in the derived class can be declared with the same name as a field in the base class. Such a field shadows the field in the base class in that it is always referenced when it is accessed in the context of the derived class. There is an expectation that there will be a way to reference the field in the base class but this is not defined at this time.

15.8.5 Overriding Base Class Methods

If a method in a derived class is declared with the identical signature as a method in a base class, then it is said to override the base class’s method. Such a method is a candidate for dynamic dispatch in the event that a variable that has the base class type references a variable that has the derived class type.

The identical signature requires that the names, types, and order of the formal arguments be identical. The return type of the overriding method must be the same as the return type of the base class’s method, or must be a subclass of the base class method’s return type.

Methods without parentheses are not candidates for dynamic dispatch.

Rationale. Methods without parentheses are primarily used for field accessors of which a default is created if none is specified. The field accessor should not dynamically dispatch in general since that would make it impossible to access a base field within a base method should that field be shadowed by a subclass.

15.8.6 Inheriting from Multiple Classes

A class can be derived from multiple base classes provided that only one of the base classes contains fields either directly or from base classes that it is derived from. The methods defined by the other base classes can be overridden.

15.9 Nested Classes

A class defined within another class is a nested class.

Nested classes can refer to fields and methods in the outer class implicitly or explicitly with an `outer` reference.

15.10 The `nil` Value

Chapel provides `nil` to indicate the absence of a reference to any object. `nil` can be assigned to a variable of any class type. Invoking a class method or accessing a field of the `nil` value results in a run-time error.

15.11 Dynamic Memory Management

Memory associated with class instances can be reclaimed with the `delete` keyword.

Example. The following example allocates a new object `c` of class type `C` and then deletes it.

```
var c = new C();
delete c;
```

Open issue. Chapel was originally specified without a `delete` keyword. The intention was that Chapel would be implemented with a distributed-memory garbage collector. This is a research challenge. In order to focus elsewhere, the design has been scaled back. There is an expectation that Chapel will eventually support an optional distributed-memory garbage collector as well as a region-based memory management scheme similar to that used in the Titanium language. Support of `delete` will likely continue even as these optional features become supported.

16 Records

A record is a data structure that is like a class but has value semantics. Like classes, records can be generic (§22.3). The key differences between records and classes are listed below.

16.1 Differences between Classes and Records

16.1.1 References vs. Values

The main difference between records and classes is that records are value classes. Record instances are manipulated as values, in the same manner as values of primitive types. Records are assigned by value; see §16.1.6 for more details. Records are also passed by value to functions, unless argument intents (§13.5) are used.

Example. The following example defines and manipulates a simple record.

```
record MyColor {
  var color: int;
}

def printMyColor(mc: MyColor) {
  writeln("my color is ", mc.color);
  mc.color = 6;   // does not affect the caller's record
}

var mc1: MyColor;         // 'color' defaults to 0
var mc2: MyColor = mc1;   // mc1's value is copied into mc2
mc1.color = 3;            // mc1's value is modified
printMyColor(mc2);        // mc2 is not affected by assignment to mc1
printMyColor(mc2);        // ... or by assignment in printMyColor()

def modifyMyColor(inout mc: MyColor, newcolor: int) {
  mc.color = newcolor;
}
modifyMyColor(mc2, 7);    // mc2 is affected because of the 'inout' intent
printMyColor(mc2);
```

The assignment to `mc1.color` affects only the record stored in `mc1`. The record in `mc2` is not affected by the assignment to `mc1` or by the assignment in `printMyColor`. `mc2` is affected by the assignment in `modifyMyColor` because the intent `inout` is used.

16.1.2 Storage Allocation

Storage for a record variable directly contains the data associated with the fields in the record, in the same manner as variables of primitive types directly contain the primitive values. Record storage is reclaimed when the record variable goes out of scope. No additional storage for a record is allocated or reclaimed. Field data of one variable's record is not shared with data of another variable's record.

By contrast, the memory for a class variable contains only a reference to a class instance. Storage for a class instance, including storage for the data associated with the fields in the class, is allocated and reclaimed separately from variables referencing that instance. The same class instance can be referenced by multiple variables.

16.1.3 Record Inheritance

When a record is derived from a base record, it contains the data in the base record. The difference between record inheritance and class inheritance is that there is no dynamic dispatch. The record type of a variable is the exact type of that variable, i.e. a variable of a base record type cannot store a derived record type.

16.1.4 No Dynamic Dispatch

Records do not support dynamic dispatch.

16.1.5 No `nil` Value

Records do not provide a counterpart of the `nil` value. A record with all fields set to their types' default values might be the closest concept to `nil`. Such a record is different, however, in that it is a legal record instance, whereas `nil` does not refer to any legal class instance.

16.1.6 Record Assignment

In record assignment, the fields of the record on the left-hand side of the assignment are assigned the corresponding field values of the record on the right-hand side of the assignment. Record assignment is generic and structural in that the right-hand side expression can be of any type as long as it contains at least the same fields (by name) as the record on the left-hand side.

A left-hand-side field must be assignable the corresponding right-hand-side field, i.e., an implicit conversion (§9.1) must exist between the fields' types. Fields on the right-hand side that do not exist on the left-hand side are ignored during record assignment. For example, when a base record is assigned a derived record, just the fields that exist in the base record are assigned. Assignment from a class instance to a record is allowed, but assignment from record to class is not.

16.2 Record Declarations

A record is defined with the following syntax:

record-declaration-statement:
  record *identifier record-inherit-list*$_{opt}$ {
    record-statement-list }

record-inherit-list:
  : *record-type-list*

record-type-list:
  record-type
  record-type , *record-type-list*

record-statement-list:
  record-statement

record-statement record-statement-list

record-statement:
type-declaration-statement
function-declaration-statement
variable-declaration-statement
empty-statement

The only difference between record and class declarations is that the `record` keyword replaces the `class` keyword.

The record type is specified as a class type is and is summarized by the following syntax:

record-type:
identifier
identifier (*named-expression-list*)

16.3 Record Construction

A variable of a record type declared without an initialization expression is initialized to a record instance by calling the record's default constructor. To construct a record instance as an expression, i.e. without binding it to a variable, the `new` keyword is required. In this case, storage is allocated and reclaimed as for a record variable declaration (§16.1.2).

Rationale. The `new` keyword disambiguates types from values. This is needed because of the close relationship between constructors and type specifiers for classes and records.

16.4 Default Comparison Operators on Records

Default functions to overload == and `!=` are defined for records if there is none defined for the record in the Chapel program. The default implementation of == applies == to each field of the two argument records and reduces the result with the `&&` operator. The default implementation of `!=` applies `!=` to each field of the two argument records and reduces the result with the `||` operator.

17 Unions

Unions have the semantics of records, however, only one field in the union can contain data at any particular point in the program's execution. Unions are safe so that an access to a field that does not contain data is a runtime error. When a union is constructed, it is in an unset state so that no field contains data.

17.1 Union Types

The syntax of a union type is summarized as follows:

union−type:
  identifier

The union type is specified by the name of the union type. This simplification from class and record types is possible because generic unions are not supported.

17.2 Union Declarations

A union is defined with the following syntax:

union−declaration−statement:
  union *identifier* { *union−statement−list* }

union−statement−list:
  union−statement
  union−statement union−statement−list

union−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

17.2.1 Union Fields

Union fields are accessed in the same way that record fields are accessed. It is a runtime error to access a field that is not currently set.

Union fields should not be specified with initialization expressions.

17.3 Union Assignment

Union assignment is by value. The field set by the union on the right-hand side of the assignment is assigned to the union on the left-hand side of the assignment and this same field is marked as set.

17.4 The Type Select Statement and Unions

The type-select statement can be applied to unions to access the fields in a safe way by determining the type of the union.

18 Ranges

Ranges represent a sequence of integral values. Ranges are either *bounded* or *unbounded*.

Bounded ranges are characterized by a low bound l, a high bound h, and a stride s. If the stride is positive, the values described by the range are $l, l+s, l+2s, l+3s, ...$ such that all of the values in the sequence are less than or equal to h. If the stride is negative, the values described by the range are $h, h+s, h+2s, h+3s, ...$ such that all of the values in the sequence are greater than or equal to l. If $l > h$, the range is considered degenerate and represents an empty sequence. Ranges support iteration over the values they represent as described in §11.8.

Unbounded ranges are those in which the low and/or high bounds are omitted. Unbounded ranges conceptually represent a countably infinite number of values.

18.1 Range Types

The type of a range is characterized by three things: (1) the type of the values being represented, (2) the boundedness of the range, and (3) whether or not the range is *stridable*.

The type of the range's values is represented using a type parameter named *idxType*. This must be one of the `int` or `uint` types. The default type is `int`.

> *Open issue*. It has been hypothesized that ranges of other types, such as floating point values, might also be of interest to represent a range of legal tolerances, for example. If you believe such support would be of interest to you, please let us know.

The boundedness of the range is represented using an enumerated parameter named *boundedType* of type `BoundedRangeType`. Legal values are `bounded`, `boundedLow`, `boundedHigh`, and `boundedNone`. The first value specifies a bounded range while the other three values specify a range in which the high bound is omitted, the low bound is omitted, or both bounds are omitted, respectively. The default value is `bounded`.

The stridability of a range is represented by a boolean parameter named *stridable*. If this parameter is set to true, the range's stride can take on any signed integer value other than 0 of the same bit-width as `idxType`. If set to false, the range's stride is fixed to 1. The default value is `false`.

> *Rationale*. The *boundedType* and *stridable* values of a range are used to optimize the generated code for common cases of ranges, as well as to optimize the implementation of domains and arrays defined using ranges.

The syntax of a range type is summarized as follows:

> *range-type*:
> **range** (*named-expression-list*)

> *Example*. The following declaration declares a variable `r` of range type that can represent ranges of 64-bit integers, with both high and low bounds specified, and the ability to have a stride other than 1.

```
var r: range(int(64), BoundedRangeType.bounded, stridable=true);
```

The default value for a range is `1..0`.

18.2 Literal Range Values

Range literals are specified as follows:

```
range-literal:
  bounded-range-literal
  unbounded-range-literal
```

18.2.1 Bounded Range Literals

A bounded range is specified by the syntax

```
bounded-range-literal:
  expression .. expression
```

The first expression is taken to be the lower bound l and the second expression is taken to be the upper bound h. The stride of the range is 1 and can be modified with the `by` operator as described in §18.4.1.

The element type of the range type is determined by the type of the low and high bound. It is either `int`, `uint`, `int(64)`, or `uint(64)`. The type is determined by conceptually adding the low and high bounds together. The boundedness of such a range is `BoundedRangeType.bounded`. The stridability of the range is `false`.

18.2.2 Unbounded Range Literals

An unbounded range is specified by the syntax

```
unbounded-range-literal:
  expression ..
  .. expression
  ..
```

The first form results in a `BoundedRangeType.boundedLow` range, the second in a `BoundedRangeType.boundedHigh` range, and the third in a `BoundedRangeType.boundedNone` range.

Unbounded ranges can be iterated over with zipper iteration (§11.8.1) and their shape conforms to the shape of the other iterators they are being iterated over with.

Example. The code

```
for i in (1..5, 3..) do
  write(i, "; ");
```

produces the output “(1, 3); (2, 4); (3, 5); (4, 6); (5, 7); ”.

It is an error to iterate over a `BoundedRangeType.boundedNone` range, a `BoundedRangeType.boundedLow` range with negative stride or a `BoundedRangeType.boundedHigh` range with positive stride.

Unbounded ranges can also be used to index into ranges, domains, arrays, and strings. In these cases, omitted bounds are inherited from the bounds of the expression being indexed.

18.3 Range Assignment

Assigning one range to another results in its low, high, and stride values being copied from the source range to the destination range.

In order for range assignment to be legal, the element type of the source range must be implicitly coercible to the element type of the destination range. The two range types must have the same boundedness parameter. It is legal to assign a non-stridable range to a stridable range, but illegal to assign a stridable range to a non-stridable range unless the stridable range has a stride value of 1.

18.4 Range Operators

18.4.1 By Operator

The `by` operator can be applied to any range to create a strided range.

The `by` operator takes a range and an integer value to yield a new range that is strided by the integer. Striding a strided range results in a stride whose value is the product of the two strides. The stride argument can either be of type `idxType` or some other integer value that can coerce to a signed integer value of the same bit-width as `idxType`.

Example. In the following declarations, range `r1` represents the odd integers between 1 and 20. Range `r2` strides `r1` by two and represents every other odd integer between 1 and 20: 1, 5, 9, ...

```
var r1 = 1..20 by 2;
var r2 = r1 by 2;
```

Rationale. *Why isn't the high bound specified first if the stride is negative?* The reason for this choice is that the `by` operator is binary, not ternary. Given a range `R` initialized to `1..3`, we want `R by -1` to contain the ordered sequence $3, 2, 1$. But then `R by -1` would be different than `3..1 by -1` even though it should be identical by substituting the value in R into the expression.

18.4.2 Count Operator

The `#` operator can be applied to a range that has a high bound, a low bound, or both.

The `#` operator takes a range and an integral count and creates a new range with *count* elements. The stride of the resulting range is the same as that of the initial range. It is an error for the count to be negative. The *idxType* of the resulting range is the same type that would be obtained by adding the integral count value to a value with the range's *idxType*.

When applied to a `BoundedRangeType.bounded` range with a positive stride, *count* elements are taken starting from the low bound. When the stride is negative, *count* elements are taken starting from the high bound. It is an error for *count* to be larger than the length of the range.

When applied to a `BoundedRangeType.boundedLow` range, the low bound is fixed and and the high bound is set based on the count and the absolute value of the stride.

When applied to a `BoundedRangeType.boundedHigh` range, the high bound is fixed and the low bound is set based on the count and the absolute value of the stride.

It is an error to apply the count operator to a `BoundedRangeType.boundedNone` range.

Example. The following declarations result in equivalent ranges.

```
var r1 = 2.. by -2 # 3;
var r2 = ..6 by -2 # 3;
var r3 = 0..6 by -2 # 3;
var r4 = 1..#6 by -2;
```

Each of these ranges represents the ordered set of three values: 6, 4, 2.

18.4.3 Arithmetic Operators

The following arithmetic operators are defined on ranges and integral types:

```
def +(r: range, s: integral): range
def +(s: integral, r: range): range
def -(r: range, s: integral): range
```

The + and − operators apply the scalar via the operator to the range's low and high bounds, producing a shifted version of the range. The element type of the resulting range is the type of the value that would result from an addition between the scalar value and a value with the range's element type. The bounded and stridable parameters for the result range are the same as for the input range.

Example. The following code creates a bounded, non-stridable range `r` which has an element type of `int` representing the values $0, 1, 2, 3$. It then uses the `+` operator to create a second range `r2` representing the values $1, 2, 3, 4$. The `r2` range is bounded, non-stridable, and represents values of type `int`.

```
var r = 0..3;
var r2 = r + 1;
```

18.4.4 Range Slicing

Ranges can be *sliced* using other ranges to create new sub-ranges. The resulting range represents the intersection between the two ranges. Range slicing is defined by using the range as a function in a call expression where the argument is another range. If the slicing range is unbounded in one or both directions, it inherits its missing bounds from the range being sliced.

> *Example*. In the following example, `r` represents the integers from 1 to 20 inclusive. Ranges `r2` and `r3` are defined using range slices and represent the indices from 3 to 20 and the odd integers between 1 and 20 respectively. Range `r4` represents the odd integers between 1 and 20 that are also divisible by 3.

```
var r = 1..20;
var r2 = r[3..];
var r3 = r[1.. by 2];
var r4 = r3[0.. by 3];
```

18.5 Predefined Functions and Methods on Ranges

def *range*`.idxType` **type**

Returns the index type of the range.

def *range*`.stridable` **type**

Returns true if the range is stridable, false otherwise.

def *range*`.boundedType` **type**

Returns boundedType of the range.

def *range*`.low : idxType`

Returns the low bound of the range.

def *range*`.high : idxType`

Returns the high bound of the range.

def *range*`.stride :` **int**`(numBits(idxType))`

Returns the stride of the range.

def *range*`.length : idxType`

Returns the number of elements in the range.

def *range*`.member(i: idxType):` **bool**

Returns whether or not `i` is in the range.

def *range*`.member(other:` **range**`):` **bool**

Returns whether or not every element in other is also in this.

def *range*.indexOrder(i: idxType): idxType

If `i` is a member of the range, returns an integer giving the ordinal value of `i` within the range using 0-based indexing. Otherwise, it returns `(-1):idxType`.

Example. The following calls show the order of index 4 in each of the given ranges:

```
(0..10).indexOrder(4) == 4
(1..10).indexOrder(4) == 3
(3..5).indexOrder(4) == 1
(0..10 by 2).indexOrder(4) == 2
(3..5 by 2).indexOrder(4) == -1
```

19 Domains

A *domain* is a first-class representation of an index set. Domains are used to specify iteration spaces, to define the size and shape of arrays (§20), and to specify aggregate operations like slicing. The indices described by a domain may be regular and structured or they may be irregular and unstructured. Chapel also supports the ability to create *subdomains* and *sparse subdomains* to represent subsets of a domain's index set. A domain's indices may potentially be distributed across multiple locales as described in §27, supporting global-view data structures.

19.1 Domain Taxonomy

This section describes Chapel's taxonomy of domain types.

19.1.1 Root Domains and Subdomains

A domain is either a *root domain* or a *subdomain*. This is represented as follows:

```
domain-type:
  root-domain-type
  subdomain-type
```

A root domain has no parent domain and can represent an arbitrary set of indices of its index type. A subdomain has an associated parent domain value and is constrained to only store indices that are also described by its parent domain.

19.1.2 Regular and Irregular Domain Types

Domain types can be thought of as falling into two major categories: regular and irregular. This is represented for root domain types as follows:

```
root-domain-type:
  regular-domain-type
  irregular-domain-type
```

Regular domains, known as *arithmetic domains*, describe multidimensional rectangular index sets. They are characterized by a tensor product of ranges and represent indices that are tuples of an integral type. Regular domains can be represented using $O(1)$ space. They are useful for representing multidimensional rectangular index sets and arrays.

An irregular domain can store an arbitrary set of indices of an arbitrary but homogenous index type. Irregular domains typically require space proportional to the number of indices being represented.

The two major classes of irregular domains in Chapel are associative domains and opaque domains:

```
irregular-domain-type:
  associative-domain-type
  opaque-domain-type
```

Associative domains represent an arbitrary set of indices of a given type and can be used to describe sets or to create dictionary-style arrays. Opaque domains are those for which the indices have no inherent names and are therefore anonymous. They can be used for representing sets and for building unstructured arrays, similar to pointer-based data structures in conventional languages.

Sparse subdomains, described in §19.6, are also considered to be irregular domains. A non-sparse subdomain inherits the regularity or irregularity of its parent domain.

19.2 Domain Characteristics

19.2.1 Domain Types

All domain types are characterized by the type of indices that they store (see §19.7). The way in which these index types are specified in the domain's type signature varies across domain types. It is defined for root domain types in §19.3 and for subdomains in §19.5.1.

19.2.2 Domain Values

A domain's value is the index set that it represents. A domain's index set can be considered either *ordered* or *unordered*, indicating whether or not there is a well-defined order defined for its indices for the purposes of things like serial iteration and I/O. The domain values for the root domain types are defined in §19.2.2. The domain values for sparse subdomain types are defined in §19.6.2.

19.2.3 Domain Identity

In addition to storing a value, domain variables have an identity that distinguishes them from other domains of the same type with the same value. This identity is used to define the domain's relationship with subdomains (§19.5), index types (§19.7), and arrays (§20.9). The identity of a domain is represented by its name.

> *Open issue*. In the future, it is likely that we will support a means of creating domain aliases, much as we support array aliases currently.

19.2.4 Runtime Representation of Domain Values

While domains are a high-level abstraction, users have control over the runtime representation of a domain's index set through Chapel's support for domain maps (§27), both standard (§32 and §31) and user-defined (§29). Chapel implementations should also document their choice of implicit domain maps (used to implement domains with no domain map specifiers).

19.3 Root Domain Types

19.3.1 Arithmetic Domain Types

Arithmetic domain types are parameterized by three things:

- `rank`, a positive `int` value indicating the number of dimensions that the domain represents;
- `idxType`, a type member representing the index type for each dimension. If unspecified, `idxType` defaults to `int`.
- `stridable`, a `bool` value indicating whether or not any of the domain's dimensions will be characterized by a strided range. If unspecified, `stridable` defaults to `false`.

If `rank` is 1, the index type represented by an arithmetic domain is `idxType`. Otherwise, the index type is the homogenous tuple type `rank*idxType`.

The syntax of an arithmetic domain type is summarized as follows:

regular-domain-type:
  domain (*named-expression-list*)

where *named-expression-list* permits the values of `rank`, `idxType`, and `stridable` to be specified using standard function call syntax.

19.3.2 Associative Domain Types

An associative domain type is parameterized by `idxType`, the type of the index that it stores. The syntax is as follows:

associative-domain-type:
  domain (*associative-index-type*)

associative-index-type:
  type-specifier

If the *associative-index-type* is an enumerated type, the associative domain is called an *enumerated domain type*—a variant of associative domain types that has some distinct characteristics, described in subsequent sections.

19.3.3 Opaque Domain Types

An opaque domain type is parameterized by the type `opaque`, indicating that the index values are anonymous and have no obvious representational name or value. The opaque domain type is given by the following syntax:

opaque-domain-type:
  domain (**opaque**)

19.4 Root Domain Values

This section describes the values, literal formats (if applicable), and default values for each root domain type.

19.4.1 Arithmetic Domain Values

An arithmetic domain's value is represented as `rank` ranges of type `range(idxType, BoundedRangeType.bounded, stridable)`. The index set for a rank 1 domain is the set of indices described by its singleton range. The index set for a rank n domain is the set of all `n*idxType` tuples described by the tensor product of its ranges. Arithmetic domain indices are ordered according to the lexicographic order of their values.

Literal arithmetic domain values are represented by a comma-separated list of range expressions of matching `idxType` enclosed in square brackets:

```
domain-literal:
  [ range-expression-list ]

range-expression-list:
  range-expression
  range-expression, range-expression-list

range-expression:
  expression
```

The type of an arithmetic domain literal is defined as follows:

- `rank` = the number of range expressions in the literal
- `idxType` = the type of the range expressions
- `stridable` = `true` if any of the range expressions are stridable, otherwise `false`

Example. The expression `[1..5, 1..5]` defines an arithmetic domain with type `domain(rank=2, idxType=int, strided=false)`.

Example. The expression `[1..5, 1..5]` defines a 5×5 arithmetic domain with the indices $(1,1),(1,2),\ldots,(1,5),(2,1),\ldots(5,5)$.

Example. In the code

```
var D: domain(2) = [1..n, 1..n];
```

`D` is defined as a two-dimensional, nonstridable arithmetic domain with an index type of `2*int` and is initialized to contain the set of indices (i,j) for all i and j such that $i \in 1,2,\ldots,n$ and $j \in 1,2,\ldots,n$.

The default value of a range type is the `rank` default range values for type `range(idxType, BoundedRangeType.bounded, stridable)`.

19.4.2 Associative Domain Values

An associative domain's value is simply the set of all index values that the domain describes. The indices of an associative domain are typically unordered. The only exception is associative domains over enumerated types which are ordered according to the order in which the enumeration's identifiers were declared.

There is no literal syntax for an associative domain, though a tuple of values of type `idxType` can be used to initialize a variable of associative domain type.

The default value for an associative domain is the empty set unless `idxType` is an enumerated type in which case the default value is the set of all identifiers in the enumeration.

> *Rationale*.
>
> The decision to have enumerated domains start fully populated was based on the observation that enumerations have a finite, typically small number of elements and that it would be common to declare arrays with values corresponding to each identifier in the enumeration. Furthermore, we considered it simpler to clear a fully-populated domain than to fully populate an empty one.
>
> In addition, we believe that fully-populated constant enumerated domains are an important case for compiler optimizations, particularly if the numeric values of the enumeration are consecutive.

19.4.3 Opaque Domain Values

An opaque domain's value is simply the unordered set of anonymous indices that the domain describes.

There is no literal syntax for an opaque domain due to the fact that the indices have no inherent names.

The default value for an opaque domain is the empty set.

19.5 Subdomains

A subdomain is a domain whose indices are guaranteed to be a subset of those described by another domain known as its *parent domain*. Subdomains have the same type as their parent domain, and by default they inherit the domain map of their parent domain. All domain types support subdomains.

> *Rationale*. Subdomains are provided in Chapel for a number of reasons: to facilitate the ability of the compiler or a reader to reason about the inter-relationship of distinct domain variables; to support the author's ability to omit redundant domain mapping specifications; to support the compiler's ability to reason about the relative alignment of multiple domains; and to improve the compiler's ability to prove away bounds checks for array accesses.

19.5.1 Subdomain Types

A subdomain type is specified using the following syntax:

subdomain−type:
  sparse$_{opt}$ **subdomain** (*domain−expression*)

This declares that *domain−expression* is the parent domain of this subdomain type. The subdomain type has the same type as its parent domain. By default it will share the parent domain's domain map. The optional *sparse* keyword permits the ability to create sparse subdomains described in §19.6.

Open issue.

An open semantic issue for subdomains is when a subdomain's subset property should be re-verified when its parent domain is reassigned and whether this should be done aggressively or lazily.

19.6 Sparse Subdomains

Sparse subdomains are irregular domains that describe an arbitrary subset of a domain, even if the parent domain is a regular domain. Sparse subdomains are useful in Chapel for defining *sparse arrays* in which a single element value occurs frequently enough that it is worthwhile to avoid storing it redundantly. The difference between a sparse subdomain's index set and its parent domain's describes the set of indices for which the sparse array will store this replicated value. See §20.8 for details about sparse arrays.

19.6.1 Sparse Domain Types

Each root domain type has a unique corresponding sparse subdomain type. Sparse subdomains whose parent domains are also sparse subdomains share the same type.

19.6.2 Sparse Domain Values

A sparse subdomain's value is simply the set of all index values that the domain describes. If the parent domain's indices were ordered, the sparse subdomain's are as well.

There is no literal syntax for an associative domain, though for a domain `D`, a tuple of values of type `index(D)` can be used to initialize a variable of sparse domain type.

The default value for a sparse subdomain value is the empty set.

Example. The following code declares a two-dimensional dense domain `D`, followed by a two dimensional sparse subdomain of `D` named `SpsD`. Since `SpsD` is uninitialized, it will initially describe the empty set of indices from `D`.

```
const D: domain(2) = [1..n, 1..n];
var SpsD: sparse subdomain(D);
```

19.7 Index Types

Each domain value has a corresponding compiler-provided *index type* which can be used to represent values belonging to that domain's index set. Index types are described using the following syntax:

index-type:
  index (*domain-expression*)

Rationale.

Index types are included in Chapel with two goals in mind. The first is to improve readability of the Chapel program by declaring variables to be members of specific domains with the intention of giving them more semantic meaning to a reader as compared to, say, storing all indices as `int` types where the semantic meanings blur.

The second goal is to provide the compiler with the ability to prove away bounds checks by giving the user the capability to assure the compiler that a given variable belongs to a particular domain and is therefore in bounds for its arrays and its parent domains' arrays.

Since index types are known to be legal for a given domain, they may also afford the opportunity to represent an index using an optimized format that doesn't simply store the index variable's value in order to support accelerated access to arrays declared over that domain. For both this reason and the previous, it may be less expensive to index into arrays using index type variables of their domains or subdomains.

Open issue.

An open issue for index types is what the semantics should be for an index type value that is live across a modification to its domain's index set—particularly one that shrinks the index set. Our hypothesis is that most stored indices will either have short lifespans or belong to constant or monotonically growing domains. But these semantics need to be defined nevertheless.

19.8 Domain Assignment

All domain types support domain assignment. Domain assignment is by value and causes the target domain variable to take on the index set of the right-hand side expression. In practice, the right-hand side expression is often another domain value; a tuple of ranges (for regular domains); or a tuple of indices or a loop that enumerates indices (for irregular domains). If the domain variable being assigned was used to declare arrays, these arrays are reallocated as discussed in §20.9.

Example. The following three assignments show ways of assigning indices to a sparse domain, `SpsD`. The first assigns the domain two index values, `(1,1)` and `(n,n)`. The second assigns the domain all of the indices along the diagonal from `(1,1)...(n,n)`. The third invokes an iterator that is written to `yield` indices read from a file named "inds.dat". Each of these assignments has the effect of replacing the previous index set with a completely new set of values.

```
SpsD = ((1,1), (n,n));
SpsD = [i in 1..n] (i,i);
SpsD = readIndicesFromFile("inds.dat");
```

19.9 Domain Index Set Manipulation

19.9.1 Querying Index Set Membership

Every domain type supports a `member(i)` method that returns a boolean value indicating whether or not the given index `i` is a member of the domain's index set.

19.9.2 Clearing a Domain's Index Set

Every domain type supports a `clear()` method that resets a domain's index set to its default value as specified in §19.4.

> *Example*. The following call will cause the sparse domain `SpsD` to describe an empty set of indices as it was when initially declared.
>
> ```
> SpsD.clear();
> ```

> *Example*. The following call causes the associative domain `HashD` to describe an empty set of indices as it did when it was initially declared.
>
> ```
> HashD.clear();
> ```

19.9.3 Adding and Removing Domain Indices

All irregular domain types support the ability to incrementally add and remove indices from their index sets. This can either be done using `add(i:idxType)` and `remove(i:idxType)` methods on a domain variable or by using the += and -= assignment operators. It is legal to add the same index to an irregular domain's index set twice, but illegal to remove an index that does not belong to the domain's index set.

As with normal domain assignments, arrays declared in terms of a domain being modified in this way will be reallocated as discussed in §20.9.

19.10 Iteration over Domains

All domains support iteration via standard for, forall, and coforall loops. These loops iterate over all of the indices that the domain describes. The type of the iterator variable for an iteration over a domain named `D` is that domain's index type, `index(D)`. If the domain's indices are ordered, a for loop will traverse the indices in order.

19.11 Slicing

In Chapel, slicing is the application of an index set to another variable using either parentheses or square brackets.

19.11.1 Domain-based Slicing

The index set used to express a slice can be represented using a domain value.

Slicing an array results in an alias to a subset of the array's elements as described in §20.6.

Slicing a domain evaluates to a new domain value whose index set is the intersection of the domain's index set and the slicing index set. The type and domain map of the result match the domain being sliced.

19.11.2 Range-based Slicing

When slicing arithmetic domains or arrays, the slice can be expressed as a list of `rank` ranges. These ranges can either be bounded or unbounded. When unbounded, they inherit their bounds from the domain or array being sliced.

> *Example*. The following code declares a two dimensional arithmetic domain `D`, and then a number of subdomains of `D` by slicing into `D` using bounded and unbounded ranges. The `InnerD` domain describes the inner indices of D, `Col2OfD` describes the 2nd column of `D`, and `AllButLastRow` describes all of `D` except for the last row.
>
> ```
> const D: domain(2) = [1..n, 1..n],
>       InnerD = D[2..n-1, 2..n-1],
>       Col2OfD = D[.., 2..2],
>       AllButLastRow = D[..n-1, ..];
> ```

> *Open issue*. For slices that use a list of ranges, our intention is to use zipper semantics vs. tensor semantics when evaluating the ranges depending on whether square brackets or parentheses are used. Currently all slices are defined using tensor semantics for simplicity. Since this may change in the future, we recommend using square brackets to express array-based slicing.

19.11.3 Rank-Change Slicing

For multidimensional arithmetic domains and arrays, substituting integral values for one or more of the ranges in a range-based slice will result in domain or array of lower rank.

The result of a rank-change slice on an array is an alias to a subset of the array's elements as described in §20.6.1.

The result of rank-change slice on a domain is a subdomain of the domain being sliced, as described in §19.5. The resulting subdomain's type will be the same as the original domain, but with a `rank` equal to the number of dimensions that were sliced by ranges rather than integers.

19.12 Domain Arguments to Functions

This section describes the semantics of passing domains as arguments to functions.

19.12.1 Formal Arguments of Domain Type

When a domain value is passed to a formal argument of compatible domain type by blank intent, it is passed by reference in order to preserve the domain's identity.

19.12.2 Domain Promotion of Scalar Functions

Domain values may be passed to a scalar function argument whose type matches the domain's index type. This results in a promotion of the scalar function as defined in §25.4.

> *Example*. Given a function `foo()` that accepts real floating point values and an associative domain `D` of type `domain(real)`, `foo` can be called with `D` as its actual argument which will result in the function being invoked for each value in the index set of `D`.

> *Example*. Given an array `A` with element type `int` declared over a one-dimensional domain `D` with `idxType int`, the array elements can be assigned their corresponding index values by writing:
>
> ```
> A = D;
> ```
>
> This is equivalent to:
>
> ```
> forall (a,i) in (A,D) do
>   a = i;
> ```

19.13 Domain Operators

19.13.1 By Operator

The `by` operator can be applied to an arithmetic domain value in order to create a strided arithmetic domain value. The right-hand operand to the `by` operator can either be an integral value or an integral tuple whose size matches the domain's rank.

The type of the resulting domain is the same as the original domain but with `stridable` set to true. In the case of an integer stride value, the value of the resulting domain is computed by applying the integer value to each range in the value using the `by` operator. In the case of a tuple stride value, the resulting domain's value is computed by applying each tuple component to the corresponding range using the `by` operator.

19.14 Predefined Functions and Methods on Domains

def *Domain*.`numIndices: dim_type`

Returns the number of indices in the domain.

19.14.1 Predefined Functions and Methods on Arithmetic Domains

def *Domain*.`dim(d: `**int**`): `**range**

Returns the range of indices described by dimension `d` of the domain.

Example. In the code

```
for i in D.dim(1) do
  for j in D.dim(2) do
    writeln(A(i,j));
```

domain `D` is iterated over by two nested loops. The first dimension of `D` is iterated over in the outer loop. The second dimension is iterated over in the inner loop.

def *Domain*.`rank` **param** `:` **int**

Returns the rank of the domain as a parameter int.

def *Domain*.`stridable` **param** `:` **bool**

Returns whether or not the domain is stridable as a parameter bool.

def *Domain*.`low:` **index**(*Domain*)

Returns the low index of the domain as a value of the domain's index type.

def *Domain*.`high:` **index**(*Domain*)

Returns the high index of the domain as a value of the domain's index type.

def *Domain*.`stride:` **int**`(numBits(idxType))` **where** `rank == 1`
def *Domain*.`stride: rank*`**int**`(numBits(idxType))`

Returns the stride of the domain as the domain's stride type (for 1D domains) or a tuple of the domain's stride type (for multidimensional domains).

def *Domain*.`translate(off: integral):` **domain**
def *Domain*.`translate(off: rank*integral):` **domain**

Returns a new domain that is the current domain translated by `off` or `off(d)` for each dimension `d`.

def *Domain*.`expand(off: integral):` **domain**
def *Domain*.`expand(off: rank*integral):` **domain**

Returns a new domain that is the current domain expanded in dimension `d` if `off` or `off(d)` is positive or contracted in dimension `d` if `off` or `off(d)` is negative.

def *Domain*.`exterior(off: integral):` **domain**
def *Domain*.`exterior(off: rank*integral):` **domain**

Returns a new domain that is the exterior portion of the current domain with `off` or `off(d)` indices for each dimension `d`. If `off` or `off(d)` is negative, compute the exterior from the low bound of the dimension; if positive, compute the exterior from the high bound.

def *Domain*.`interior(off: integral):` **domain**
def *Domain*.`interior(off: rank*integral):` **domain**

Returns a new domain that is the interior portion of the current domain with `off` or `off(d)` indices for each dimension `d`. If `off` or `off(d)` is negative, compute the interior from the low bound of the dimension; if positive, compute the interior from the high bound.

20 Arrays

An *array* is a map from a domain's indices to a collection of variables of homogenous type. Since Chapel domains support a rich variety of index sets, Chapel arrays are also richer than the traditional linear or rectilinear array types in conventional languages. Like domains, arrays may be distributed across multiple locales without explicitly partitioning them using Chapel's Domain Maps (§27).

20.1 Array Types

An array type is specified by the identity of the domain that it is declared over and the element type of the array. Array types are given by the following syntax:

```
array-type:
  [ domain-expression ] type-specifier

domain-expression:
  domain-literal
  expression
```

The *domain-expression* must specify a domain that the array can be declared over. This can be a domain literal. If it is a domain literal, the duplicate square brackets around the domain literal can be omitted.

Example. In the code

```
const D: domain(2) = [1..10, 1..10];
var A: [D] real;
```

`A` is declared to be an arithmetic array over arithmetic domain `D` with elements of type `real`. As a result, it represents a 2-dimensional 10×10 real floating point variables indexed using the indices $(1,1),(1,2),\ldots,(1,10),(2,1),\ldots,(10,10)$.

An array's element type can be referred to using the member symbol `eltType`.

Example. In the following example, `x` is declared to be of type `real` since that is the element type of array `A`.

```
var A: [D] real;
var x: A.eltType;
```

20.2 Array Values

An array's value is the collection of its elements' values. Assignments between array variables are performed by value as described in §20.5. Chapel semantics are defined so that the compiler will never need to insert temporary arrays of the same size as a user array variable.

Arrays do not have a literal format in Chapel, but arithmetic array variables can be initialized using tuple values. These tuples must match the size and shape of the array itself.

Example. The following example declares a 2×3 array `A` using an anonymous domain value and initializes the elements of the array using a 2-tuple of 3-tuples which matches the array's size and shape.

```
var A: [1..2, 1..3] real = ((1.1, 1.2, 1.3), (2.1, 2.2, 2.3));
```

An array's default value is to have its elements all initialized to the default values for their types.

20.2.1 Runtime Representation of Array Values

The runtime representation of an array in memory is controlled by its domain's domain map. Through this mechanism, users can reason about and control the runtime representation of an array's elements. See §27 for more details.

20.3 Array Indexing

Arrays can be indexed using index values from the domain over which they are declared. Array indexing is expressed using either parenthesis or square brackets. This results in a reference to the element that corresponds to the index value.

Example. Given:

```
var A: [1..10] real;
```

the first two elements of A can be assigned the value 1.2 and 3.4 respectively using the assignment:

```
A(1) = 1.2;
A[2] = 3.4;
```

If an array is indexed using an index that is not part of its domain's index set, the reference is considered out-of-bounds and a runtime error will occur, halting the program.

20.3.1 Arithmetic Array Indexing

Since the indices for multidimensional arithmetic domains are tuples, for convenience, arithmetic arrays can be indexed using the list of integer values that make up the tuple index. This is semantically equivalent to creating a tuple value out of the integer values and using that tuple value to index the array. For symmetry, 1-dimensional arithmetic arrays can be accessed using 1-tuple indices even though their index type is an integral value. This is semantically equivalent to de-tupling the integral value from the 1-tuple and using it to index the array.

Example. Given:

```
var A: [1..5, 1..5] real;
var ij: 2*int = (1, 1);
```

the elements of array A can be indexed using any of the following idioms:

```
A(ij) = 1.1;
A((1, 2)) = 1.2;
A(1, 3) = 1.3;
A[ij] = -1.1;
A[(1, 4)] = 1.4;
A[1, 5] = 1.5;
```

Example. The code

```
def f(A: [], is...)
  return A(is);
```

defines a function that takes an array as the first argument and a variable-length argument list. It then indexes into the array using the tuple that captures the actual arguments. This function works even for one-dimensional arrays because one-dimensional arrays can be indexed into by 1-tuples.

20.4 Iteration over Arrays

All arrays support iteration via for, forall and coforall loops. These loops iterate over all of the array elements as described by its domain. A loop of the form:

```
[co]for[all] a in A
  ...a...
```

is semantically equivalent to:

```
[co]for[all] i in A.domain
  ...A(i)...
```

Thus, the iterator variable for an array traversal is a reference to the array element type.

20.5 Array Assignment

Array assignment is by value. Arrays can be assigned arrays, ranges, domains, iterators, or tuples. If `A` is an lvalue of array type and `B` is an expression of either array, range, or domain type, or an iterator, then the assignment

```
A = B;
```

is equivalent to

```
forall (a,b) in (A,B) do
  a = b;
```

If the zipper iteration is illegal, then the assignment is illegal. Notice that the assignment is implemented with the semantics of a `forall` loop.

Arrays can be assigned tuples of values of their element type if the tuple contains the same number of elements as the array. For multidimensional arrays, the tuple must be a nested tuple such that the nesting depth is equal to the rank of the array and the shape of this nested tuple must match the shape of the array. The values are assigned element-wise.

Arrays can also be assigned single values of their element type. In this case, each element in the array is assigned this value. If `e` is an expression of the element type of the array or a type that can be implicitly converted to the element type of the array, then the assignment

```
A = e;
```

is equivalent to

```
forall a in A do
  a = e;
```

20.6 Array Slicing

An array can be sliced using a domain that has the same type as the domain over which it was declared. The result of an array slice is an alias to the subset of the array elements from the original array corresponding to the slicing domain's index set.

Example. Given the definitions

```
var OuterD: domain(2) = [0..n+1, 0..n+1];
var InnerD: domain(2) = [1..n, 1..n];
var A, B: [OuterD] real;
```

the assignment given by

```
A[InnerD] = B[InnerD];
```

assigns the elements in the interior of `B` to the elements in the interior of `A`.

20.6.1 Arithmetic Array Slicing

An arithmetic array can be sliced by any arithmetic domain that is a subdomain of the array's defining domain. If the subdomain relationship is not met, an out-of-bounds error will occur. The result is a subarray whose indices are those of the slicing domain and whose elements are an alias of the original array's.

Arithmetic arrays also support slicing by ranges directly. If each dimension is indexed by a range, this is equivalent to slicing the array by the arithmetic domain defined by those ranges. These range-based slices may also be expressed using partially unbounded or completely unbounded ranges. This is equivalent to slicing the array's defining domain by the specified ranges to create a subdomain as described in §19.11 and then using that subdomain to slice the array.

For multidimensional arithmetic arrays, slicing with a rank change is supported by substituting integral values within a dimension's range for an actual range. The resulting array will have a rank less than the arithmetic array's rank and equal to the number of ranges that are passed in to take the slice.

Example. Given an array

```
var A: [1..n, 1..n] int;
```

the slice `A[1..n, 1]` is a one-dimensional array whose elements are the first column of `A`.

20.7 Array Arguments to Functions

Arrays are passed to functions by reference. Formal arguments that receive arrays are aliases of the actual arguments.

When a formal argument has array type, the element type of the array can be omitted and/or the domain of the array can be queried or omitted. In such cases, the argument is generic and is discussed in §22.1.6.

If a non-queried domain is specified in the array type of a formal argument, the domain must match the domain of the actual argument. This is verified at runtime. There is an exception if the domain is an arithmetic domain, described in §20.7.1.

20.7.1 Formal Arguments of Arithmetic Array Type

Formal arguments of arithmetic array type allow an arithmetic domain to be specified that does not match the arithmetic domain of the actual arithmetic array that is passed to the formal argument. In this case, the shape (size in each dimension and rank) of the domain of the actual array must match the shape of the domain of the formal array. The indices are translated in the formal array, which is a reference to the actual array.

Example. In the code

```
def foo(X: [1..5] int) { ... }
var A: [1..10 by 2] int;
foo(A);
```

the array `A` is strided and its elements can be indexed by the odd integers between one and nine. In the function `foo`, the array `X` references array `A` and the same elements can be indexed by the integers between one and five.

20.7.2 Array Promotion of Scalar Functions

Array promotion of a scalar function is defined over the array type and the element type of the array. The domain of the returned array, if an array is captured by the promotion, is the domain of the array that promoted the function. In the event of zipper promotion over multiple arrays, the promoted function returns an array with a domain that is equal to the domain of the first argument to the function that enables promotion. If the first argument is an iterator or a range, the result is a one-based one-dimensional array.

Example. Whole array operations is a special case of array promotion of scalar functions. In the code

```
A = B + C;
```

if `A`, `B`, and `C` are arrays, this code assigns each element in `A` the element-wise sum of the elements in `B` and `C`.

20.7.3 Array Aliases

Array slices alias the data in arrays rather than copying it. Such array aliases can be captured and optionally reindexed with the array alias operator =>. The syntax for capturing an alias to an array requires a new variable declaration:

> *array−alias−declaration*:
> *identifier reindexing−expression*$_{opt}$ => *array−expression* ;
>
> *reindexing−expression*:
> [*domain−expression*]
>
> *array−expression*:
> *expression*

The identifier is an alias to the array specified in the *array−expression*.

The optional *reindexing−expression* allows the domain of the array alias to be reindexed. The shape of the domain in the *reindexing−expression* must match the shape of the domain of the *array−expression*. Indexing via the alias is governed by the new indices.

Example. In the code

```
var A: [1..5, 1..5] int;
var AA: [0..2, 0..2] => A[2..4, 2..4];
```

an array alias `AA` is created to alias the interior of array `A` given by the slice `A[2..4, 2..4]`. The reindexing expression changes the indices defined by the domain of the alias to be zero-based in both dimensions. Thus `AA(1,1)` is equivalent to `A(3,3)`.

20.8 Sparse Arrays

Sparse arrays in Chapel are those whose domain is a sparse array. A sparse array differs from other array types in that it stores a single value corresponding to multiple indices. This value is commonly referred to as the *zero value*, but we refer to it as the *implicitly replicated value* or *IRV* since it can take on any value of the array's element type in practice including non-zero numeric values, a class reference, a record or tuple value, etc.

An array declared over a sparse domain can be indexed using any of the indices in the sparse domain's parent domain. If it is read using an index that is not part of the sparse domain's index set, the IRV value is returned. Otherwise, the array element corresponding to the index is returned.

Sparse arrays can only be written at locations corresponding to indices in their domain's index set. In general, writing to other locations corresponding to the IRV value will result in a runtime error.

By default a sparse array's IRV is defined as the default value for the array's element type. The IRV can be set to any value of the array's element type by assigning to a pseudo-field named `IRV` in the array.

Example. The following code example declares a sparse array, `SpsA` using the sparse domain `SpsD` (For this example, assume that `n`>1). Line 2 assigns two indices to `SpsD`'s index set and then lines 3–4 store the values 1.1 and 9.9 to the corresponding values of `SpsA`. The IRV of `SpsA` will initially be 0.0 since its element type is `real`. However, the fifth line sets the IRV to be the value 5.5, causing `SpsA` to represent the value 1.1 in its low corner, 9.9 in its high corner, and 5.5 everywhere else. The final statement is an error since it attempts to assign to `SpsA` at an index not described by its domain, `SpsD`.

```
var SpsA: [SpsD] real;
SpsD = ((1,1), (n,n));
SpsA(1,1) = 1.1;
SpsA(n,n) = 9.9;
SpsA.IRV = 5.5;
SpsA(1,n) = 0.0;  // ERROR!
```

20.9 Association of Arrays to Domains

When an array is declared, it is linked during execution to the domain identity over which it was declared. This linkage is invariant for the array's lifetime and cannot be changed.

When indices are added or removed from a domain, the change impacts the arrays declared over this particular domain. In the case of adding an index, an element is added to the array and initialized to the IRV for sparse arrays, and to the default value for the element type for dense arrays. In the case of removing an index, the element in the array is removed.

When a domain is reassigned a new value, its arrays are also impacted. Values that correspond to indices in the intersection of the old and new domain are preserved in the arrays. Values that could only be indexed by the old domain are lost. Values that can only be indexed by the new domain have elements added to the new array, initialized to the IRV for sparse arrays, and to the element type's default value for other array types.

For performance reasons, there is an expectation that a method will be added to domains to allow non-preserving assignment, *i.e.*, all values in the arrays associated with the assigned domain will be lost. Today this can be achieved by assigning the array's domain an empty index set (causing all array elements to be deallocated) and then re-assigning the new index set to the domain.

An array's domain can only be modified directly, via the domain's name or an alias created by passing it to a function via blank intent. In particular, the domain may not be modified via the array's `.domain` method, nor by using the domain query syntax on a function's formal array argument (§22.1.6).

Rationale. When multiple arrays are declared using a single domain, modifying the domain affects all of the arrays. Allowing an array's domain to be queried and then modified suggests that the change should only affect that array. By requiring the domain to be modified directly, the user is encouraged to think in terms of the domain distinctly from a particular array.

In addition, this choice has the beneficial effect that arrays declared via an anonymous domain have a constant domain. Constant domains are considered a common case and have potential compilation benefits such as eliminating bounds checks. Therefore making this convenient syntax support a common, optimizable case seems prudent.

20.10 Predefined Functions and Methods on Arrays

There is an expectation that this list of predefined functions and methods will grow.

def *Array*.eltType **type**

> Returns the element type of the array.

def *Array*.rank **param**

> Returns the rank of the array.

def *Array*.**domain**: this.**domain**

> Returns the domain of the given array. This domain is constant, implying that the domain cannot be resized by assigning to its domain field, only by modifying the domain directly.

def *Array*.numElements: this.**domain**.dim_type

> Returns the number of elements in the array.

def reshape(A: *Array*, D: *Domain*): *Array*

> Returns a copy of the array containing the same values but in the shape of the new domain. The number of indices in the domain must equal the number of elements in the array. The elements of the array are copied into the new array using the default iteration orders over both arrays.

21 Iterators

An iterator is a function that conceptually returns multiple values rather than simply a single value.

> *Open issue*. The parallel iterator story is under development. It is expected that the specification will be expanded regarding parallel iterators soon.

21.1 Iterator Functions

The syntax of an iterator declaration is identical to that of a function declaration. A function is an iterator if it includes yield statements. When a yield is encountered, the value is returned, but the iterator is not finished evaluating when called within a loop. It will continue from the point after the yield and can yield or return more values. When a return is encountered, the value is returned and the iterator finishes. An iterator also completes after the last statement in the iterator function is executed.

21.2 The Yield Statement

The yield statements can only appear in iterators. The syntax of the yield statement is given by

> *yield-statement*:
>   **yield** *expression* ;

21.3 Iterator Calls

Iterator functions can be called within for or forall loops, in which case they are executed in an interleaved manner with the body of the loop. An iterator function call, or iterator invocation, can be used in an expression context, in which case it evaluates to a 1-based array of values. An iterator invocation can also be passed to a generic function argument, in which case it will not be evaluted until the formal argument is referenced within the function.

21.3.1 Iterators in For and Forall Loops

When an iterator is accessed via a for or forall loop, the iterator is evaluated alongside the loop body in an interleaved manner. For each iteration, the iterator yields a value and the body is executed.

21.3.2 Iterators as Arrays

If an iterator function is captured into a new variable declaration or assigned to an array, the iterator is iterated over in total and the expression evaluates to a one-dimensional arithmetic array that contains the values returned by the iterator on each iteration.

Example. Given an iterator

```
def squares(n: int): int {
  for i in 1..n do
    yield i * i;
}
```

the expression `squares(5)` evaluates to the array `1, 4, 9, 16, 25`.

21.3.3 Iterators and Generics

An iterator call expression can be passed to a generic function argument that has neither a declared type nor default value (§22.1.3). In this case the iterator is passed without being evaluated. Within the generic function the corresponding formal argument can be used as an iterator, e.g. in for loops. The arguments to the iterator call expression, if any, are evaluated at the call site, i.e. prior to passing the iterator to the generic function.

21.3.4 Recursive Iterators

Recursive iterators are allowed. A recursive iterator invocation is typically made by iterating over it in a loop.

Example. A post-order traversal of a tree data structure could be written like this:

```
def postorder(tree: Tree): string {
  if tree != nil {
    for child in postorder(tree.left) do
      yield child;
    for child in postorder(tree.right) do
      yield child;
    yield tree.data;
  }
}
```

By contrast, using calls `postorder(tree.left)` and `postorder(tree.right)` as stand-alone statements would result in generating temporary arrays containing the outcomes of these recursive calls, which would then be discarded.

21.4 Parallel Iterators

Iterators used in explicit forall-statements or -expressions must be parallel iterators. Reductions, scans and promotion over serial iterators will be serialized.

The definition of parallel iterators is forthcoming. Parallel iterators are defined over standard constructs in Chapel such as ranges, domains, and arrays (including Block- and Cyclic-distributed domains and arrays), thereby allowing these constructs to be used with forall-statements and -expressions.

22 Generics

Chapel supports generic functions and types that are parameterizable over both types and parameters. The generic functions and types look similar to non-generic functions and types already discussed.

22.1 Generic Functions

A function is generic if any of the following conditions hold:

- Some formal argument is specified with an intent of `type` or `param`.
- Some formal argument has no specified type and no default value.
- Some formal argument is specified with a queried type.
- The type of some formal argument is a generic type, e.g., `List`. Queries may be inlined in generic types, e.g., `List(?eltType)`.
- The type of some formal argument is an array type where either the element type is queried or omitted or the domain is queried or omitted.

These conditions are discussed in the next sections.

22.1.1 Formal Type Arguments

If a formal argument is specified with intent `type`, then a type must be passed to the function at the call site. A copy of the function is instantiated for each unique type that is passed to this function at a call site. The formal argument has the semantics of a type alias.

Example. The following code defines a function that takes two types at the call site and returns a 2-tuple where the types of the components of the tuple are defined by the two type arguments and the values are specified by the types default values.

```
def build2Tuple(type t, type tt) {
  var x1: t;
  var x2: tt;
  return (x1, x2);
}
```

This function is instantiated with "normal" function call syntax where the arguments are types:

```
var t2 = build2Tuple(int, string);
t2 = (1, "hello");
```

22.1.2 Formal Parameter Arguments

If a formal argument is specified with intent `param`, then a parameter must be passed to the function at the call site. A copy of the function is instantiated for each unique parameter that is passed to this function at a call site. The formal argument is a parameter.

Example. The following code defines a function that takes an integer parameter `p` at the call site as well as a regular actual argument of integer type `x`. The function returns a homogeneous tuple of size `p` where each component in the tuple has the value of `x`.

```
def fillTuple(param p: int, x: int) {
  var result: p*int;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

The function call `fillTuple(3, 3)` returns a 3-tuple where each component contains the value `3`.

22.1.3 Formal Arguments without Types

If the type of a formal argument is omitted, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site. A copy of the function is instantiated for each unique actual type.

Example. The example from the previous section can be extended to be generic on a parameter as well as the actual argument that is passed to it by omitting the type of the formal argument `x`. The following code defines a function that returns a homogeneous tuple of size `p` where each component in the tuple is initialized to `x`:

```
def fillTuple(param p: int, x) {
  var result: p*x.type;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

In this function, the type of the tuple is taken to be the type of the actual argument. The call `fillTuple(3, 3.14)` returns a 3-tuple of real values `(3.14, 3.14, 3.14)`. The return type is `(real, real, real)`.

22.1.4 Formal Arguments with Queried Types

If the type of a formal argument is specified as a queried type, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site. A copy of the function is instantiated for each unique actual type. The queried type has the semantics of a type alias.

Example. The example from the previous section can be rewritten to use a queried type for clarity:

```
def fillTuple(param p: int, x: ?t) {
  var result: p*t;
  for param i in 1..p do
    result(i) = x;
  return result;
}
```

22.1.5 Formal Arguments of Generic Type

If the type of a formal argument is a generic type, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site with the constraint that the type of the actual argument is an instantiation of the generic type. A copy of the function is instantiated for each unique actual type.

Example. The following code defines a function `writeTop` that takes an actual argument that is a generic stack (see §22.6) and outputs the top element of the stack. The function is generic on the type of its argument.

```
def writeTop(s: Stack) {
  write(s.top.item);
}
```

Types and parameters may be queried from the top-level types of formal arguments as well. In the example above, the formal argument's type could also be specified as `Stack(?type)` in which case the symbol `type` is equivalent to `s.itemType`.

Note that generic types which have default values for all of their generic fields, *e.g. range*, are not generic when simply specified and require a query to mark the argument as generic. For simplicity, the identifier may be omitted.

Example. The following code defines a class with a type field that has a default value. Function `f` is defined to take an argument of this class type where the type field is instantiated to the default. Function `g`, on the other hand, is generic on its argument because of the use of the question mark.

```
class C {
  type t = int;
}
def f(c: C) {
  // c.type is always int
}
def g(c: C(?)) {
  // c.type may not be int
}
```

The generic type may be specified with some queries and some exact values. Thesse exact values result in *implicit where clauses* for the purpose of function resolution.

Example. Given the class definition

```
class C {
  type t;
  type tt;
}
```

then the function definition

```
def f(c: C(?t,real)) {
  // body
}
```

is equivalent to

```
def f(c: C(?t,?tt)) where tt == real {
  // body
}
```

For tuples with query arguments, an implicit where clause is always created to ensure that the size of the actual tuple matches the implicitly specified size of the formal tuple.

Example. The function definition

```
def f(tuple: (?t,real)) {
  // body
}
```

is equivalent to

```
def f(tuple: (?t,?tt)) where tuple.size == 2 && tt == real {
  // body
}
```

The generic types `integral`, `numeric` and `enumerated` are generic types that can only be instantiated with, respectively, the signed and unsigned integral types, all of the numeric types, and enumerated types.

22.1.6 Formal Arguments of Generic Array Types

If the type of a formal argument is an array where either the domain or the element type is queried or omitted, the type of the formal argument is taken to be the type of the actual argument passed to the function at the call site. If the domain is omitted, the domain of the formal argument is taken to be the domain of the actual argument.

A queried domain may not be modified via the name to which it is bound (see §20.9 for rationale).

22.2 Function Visibility in Generic Functions

Function visibility in generic functions is altered depending on the instantiation. When resolving function calls made within generic functions, the visible functions are taken from any call site at which the generic function is instantiated for each particular instantiation. The specific call site chosen is arbitrary and it is referred to as the *point of instantiation*.

For function calls that specify the module explicitly (§12.4.1), an implicit use of the specified module exists at the call site.

Example. Consider the following code which defines a generic function `bar`:

```
module M1 {
  record R {
    var x: int;
    def foo() { }
  }
}

module M2 {
  def bar(x) {
    x.foo();
  }
}

module M3 {
  use M1, M2;
  def main() {
    var r: R;
    bar(r);
  }
}
```

In the function `main`, the variable `r` is declared to be of type `R` defined in module `M1` and a call is made to the generic function `bar` which is defined in module `M2`. This is the only place where `bar` is called in this program and so it becomes the point of instantiation for `bar` when the argument `x` is of type `R`. Therefore, the call to the `foo` method in `bar` is resolved by looking for visible functions from within `main` and going through the use of module `M1`.

If the generic function is only called indirectly through dynamic dispatch, the point of instantiation is defined as the point at which the derived type (the type of the implicit `this` argument) is defined or instantiated (if the derived type is generic).

Rationale. Visible function lookup in Chapel's generic functions is handled differently than in C++'s template functions in that there is no split between dependent and independent types.

Also, dynamic dispatch and instantiation is handled differently. Chapel supports dynamic dispatch over methods that are generic in some of its formal arguments.

Note that the Chapel lookup mechanism is still under development and discussion. Comments or questions are appreciated.

22.3 Generic Types

Generic types are generic classes and generic records. A class or record is generic if it contains one or more generic fields. A generic field is one of:

- a specified or unspecified type alias,
- a parameter field, or
- a `var` or `const` field that has no type and no initialization expression.

For each generic field, the class or record is parameterized over:

- the type bound to the type alias,
- the value of the parameter field, or
- the type of the `var` or `const` field, respectively.

Correspondingly, the class or record is instantiated with a set of types and parameter values, one type or value per generic field.

22.3.1 Type Aliases in Generic Types

If a class or record defines a type alias, the class or record is generic over the type that is bound to that alias. Such a type alias is accessed as if it were a field; similar to a parameter field, it cannot be assigned except in its declaration.

The type alias becomes an argument with intent `type` to the default constructor (§22.3.6) for its class or record. This makes the default constructor generic. The type alias also becomes an argument with intent `type` to the type constructor (§22.3.4). If the type alias declaration binds it to a type, that type becomes the default for these arguments, otherwise they have no defaults.

The class or record is instantiated by binding the type alias to the actual type passed to the corresponding argument of a user-defined (§22.3.7) or default constructor or type constructor. If that argument has a default, the actual type can be omitted, in which case the default will be used instead.

Example. The following code defines a class called `Node` that implements a linked list data structure. It is generic over the type of the element contained in the linked list.

```
class Node {
  type eltType;
  var data: eltType;
  var next: Node(eltType);
}
```

The call `new Node(real, 3.14)` creates a node in the linked list that contains the value `3.14`. The `next` field is set to nil. The type specifier `Node` is a generic type and cannot be used to define a variable. The type specifier `Node(real)` denotes the type of the `Node` class instantiated over `real`. Note that the type of the `next` field is specified as `Node(eltType)`; the type of `next` is the same type as the type of the object that it is a field of.

22.3.2 Parameters in Generic Types

If a class or record defines a parameter field, the class or record is generic over the value that is bound to that field. The parameter becomes an argument with intent `param` to the default constructor (§22.3.6) for that class or record. This makes the default constructor generic. The parameter also becomes an argument with intent `param` to the type constructor (§22.3.4). If the parameter declaration has an initialization expression, that expression becomes the default for these arguments, otherwise they have no defaults.

The class or record is instantiated by binding the parameter to the actual value passed to the corresponding argument of a user-defined (§22.3.7) or default constructor or type constructor. If that argument has a default, the actual value can be omitted, in which case the default will be used instead.

Example. The following code defines a class called `IntegerTuple` that is generic over an integer parameter which defines the number of components in the class.

```
class IntegerTuple {
  param size: int;
  var data: size*int;
}
```

The call `new IntegerTuple(3)` creates an instance of the `IntegerTuple` class that is instantiated over parameter `3`. The field `data` becomes a 3-tuple of integers. The type of this class instance is `IntegerTuple(3)`. The type specified by `IntegerTuple` is a generic type.

22.3.3 Fields without Types

If a `var` or `const` field in a class or record has no specified type or initialization expression, the class or record is generic over the type of that field. The field becomes an argument with blank intent to the default constructor (§22.3.6). That argument has no specified type and no default value. This makes the default constructor generic. The field also becomes an argument with `type` intent and no default to the type constructor (§22.3.4). Correspondingly, an actual value must always be passed to the default constructor argument and an actual type to the type constructor argument.

The class or record is instantiated by binding the type of the field to the type of the value passed to the corresponding argument of a user-defined (§22.3.7) or default constructor. When the type constructor is invoked, the class or record is instantiated by binding the type of the field to the actual type passed to the corresponding argument.

Example. The following code defines another class called `Node` that implements a linked list data structure. It is generic over the type of the element contained in the linked list. This code does not specify the element type directly in the class as a type alias but rather omits the type from the `data` field.

```
class Node {
  var data;
  var next: Node(data.type) = nil;
}
```

A node with integer element type can be defined in the call to the constructor. The call `new Node(1)` defines a node with the value `1`. The code

```
var list = new Node(1);
list.next = new Node(2);
```

defines a two-element list with nodes containing the values `1` and `2`. The type of each object could be specified as `Node(int)`.

22.3.4 The Type Constructor

A type constructor is automatically created for each class or record. A type constructor is a type function (§13.9) that has the same name as the class or record. It takes one argument per the class's or record's generic field, including fields inherited from the superclasses, if any. The formal argument has intent `type` for a type alias field and is a parameter for a parameter field. It accepts the type to be bound to the type alias and the value to be bound to the parameter, respectively. For a generic `var` or `const` field, the corresponding formal argument also has intent `type`. It accepts the type of the field, as opposed to a value as is the case for a parameter field. The formal arguments occur in the same order as the fields are declared and have the same names as the corresponding fields. Unlike the default constructor, the type constructor has only those arguments that correspond to generic fields.

A call to a type constructor accepts actual types and parameter values and returns the type of the class or record that is instantiated appropriately for each field (§22.3.1, §22.3.2, §22.3.3). Such an instantiated type must be used as the type of a variable, array element, non-generic formal argument, and in other cases where uninstantiated generic class or record types are not allowed.

When a generic field declaration has an initialization expression or a type alias is specified, that initializer becomes the default value for the corresponding type constructor argument. Uninitialized fields, including all generic `var` and `const` fields, and unspecified type aliases result in arguments with no defaults; actual types or values for these arguments must always be provided when invoking the type constructor.

22.3.5 Generic Methods

All methods bound to generic classes or records, including constructors, are generic over the implicit `this` argument. This is in addition to being generic over any other argument that is generic.

22.3.6 The Default Constructor

The default constructor for a class or record (§15.6.2) is generic over each argument that corresponds to a generic field, as specified above. The argument has intent `type` for a type alias field and is a parameter for a parameter field. It accepts the type to be bound to the type alias and the value to be bound to the parameter, respectively. This is the same as for the type constructor. For a generic `var` or `const` field, the corresponding formal argument has the blank intent and accepts the value for the field to be initialized with. The type of the field is inferred automatically to be the type of the initialization value.

The default values for the generic arguments of the default constructor are the same as for the type constructor (§22.3.4). For example, the arguments corresponding to the generic `var` and `const` fields, if any, never have defaults, so the corresponding actual values must always be provided.

22.3.7 User-Defined Constructors

If a generic field of a class does not have an initialization expression or a type alias is unspecified, each user-defined constructor for that class must provide a formal argument whose name matches the name of the field.

If the name of a formal argument in a user-defined constructor matches the name of a generic field that does not have an initialization expression, is a type alias, or is a parameter field, the field is automatically initialized at the beginning of the constructor invocation to the actual value of that argument. This is done by passing that formal argument to the implicit invocation of the default constructor at the start of the user-defined constructor (§15.6.1).

Example. In the following code:

```
class MyGenericClass {
  type t1;
  param p1;
  const c1;
  var v1;
  var x1: t1; // this field is not generic

  type t5 = real;
  param p5 = "a string";
  const c5 = 5.5;
  var v5 = 555;
  var x5: t5; // this field is not generic

  def MyGenericClass(c1, v1, type t1, param p1) { }
  def MyGenericClass(type t5, param p5, c5, v5, x5,
                     type t1, param p1, c1, v1, x1) { }
}  // class MyGenericClass

var g1 = new MyGenericClass(11, 111, int, 1);
var g2 = new MyGenericClass(int, "this is g2", 3.3, 333, 3333,
                            real, 2, 222, 222.2, 22);
```

The arguments `t1`, `p1`, `c1`, and `v1` are required in all constructors for `MyGenericClass`. They can appear in any order. Both `MyGenericClass` constructors initialize the corresponding fields implicitly because these fields do not have initialization expressions. The second constructor also initializes implicitly the fields `t5` and `p5` because they are a type field and a parameter field. It does not initialize the fields `c5` and `v5` because they have initialization expressions, or the fields `x1` and `x5` because they are not generic fields (even though they are of generic types).

Open issue. The design of constructors, especially for generic classes, is under development, so the above specification may change.

22.4 Where Expressions

The instantiation of a generic function can be constrained by *where clauses*. A where clause is specified in the definition of a function (§13.2). When a function is instantiated, the expression in the where clause must be a parameter expression and must evaluate to either `true` or `false`. If it evaluates to `false`, the instantiation is rejected and the function is not a possible candidate for function resolution. Otherwise, the function is instantiated.

Example. Given two overloaded function definitions

```
def foo(x) where x.type == int { writeln("int"); }
def foo(x) where x.type == real { writeln("real"); }
```

the call foo(3) resolves to the first definition because when the second function is instantiated the where clause evaluates to false.

22.5 User-Defined Compiler Diagnostics

The special compiler diagnostic function calls `compilerError` and `compilerWarning` generate compiler diagnostic of the indicated severity if the function containing these calls may be called when the program is executed and the function call is not eliminated by parameter folding.

The compiler diagnostic is defined by the actual arguments which must be string parameters. The diagnostic points to the spot in the Chapel program from which the function containing the call is called. Compilation halts if a `compilerError` is encountered whereas it will continue after encountering a `compilerWarning`.

Note that when a variable function is called in a context where the implicit `setter` argument is true or false, both versions of the variable function are resolved by the compiler. Consequently, the `setter` argument cannot be effectively used to guard a compiler diagnostic statements.

Example. The following code shows an example of using user-defined compiler diagnostics to generate warnings and errors:

```
def foo(x, y) {
  if (x.type != y.type) then
    compilerError("foo() called with non-matching types: ",
                  typeToString(x.type), " != ", typeToString(y.type));
  writeln("In 2-argument foo...");
}

def foo(x) {
  compilerWarning("1-argument version of foo called");
  writeln("In generic foo!");
}
```

The first routine generates a compiler error whenever the compiler encounters a call to it where the two arguments have different types. It prints out an error message indicating the types of the arguments. The second routine generates a compiler warning whenver the compiler encounters a call to it.

Thus, if the program foo.chpl contained the following calls:

```
1 foo(3.4);
2 foo("hi");
3 foo(1, 2);
4 foo(1.2, 3.4);
5 foo("hi", "bye");
6 foo(1, 2.3);
7 foo("hi", 2.3);
```

compiling the program would generate output like:

```
foo.chpl:1: warning: 1-argument version of foo called with type: real
foo.chpl:2: warning: 1-argument version of foo called with type: string
foo.chpl:6: error: foo() called with non-matching types: int != real
```

22.6 Example: A Generic Stack

```
class MyNode {
  type itemType;               // type of item
  var item: itemType;          // item in node
  var next: MyNode(itemType); // reference to next node (same type)
}

record Stack {
  type itemType;              // type of items
  var top: MyNode(itemType); // top node on stack linked list

  def push(item: itemType) {
    top = new MyNode(itemType, item, top);
  }

  def pop() {
    if isEmpty then
      halt("attempt to pop an item off an empty stack");
    var oldTop = top;
    top = top.next;
    return oldTop.item;
  }

  def isEmpty return top == nil;
}
```

23 Input and Output

Chapel provides a built-in `file` class to handle input and output to files using functions and methods called `read`, `readln`, `write`, and `writeln`.

23.1 The *file* type

The file class contains the following fields:

- The `filename` field is a `string` that contains the name of the file.
- The `path` field is a `string` that contains the path of the file.
- The `mode` field is a `FileAccessMode` enum value that indicates whether the file is being read or written.
- The `style` field can be set to `text` or `binary` to specify that reading from or writing to the file should be done with text or binary formats.

These fields can be modified any time that the file is closed.

The `mode` field supports the following `FileAccessMode` values:

- `FileAccessMode.read` The file can be read.
- `FileAccessMode.write` The file can be written.

The file type supports the following methods:

- The `open()` method opens the file for reading and/or writing.
- The `close()` method closes the file for reading and/or writing.
- The `isOpen` method returns true if the file is open for reading and/or writing, and otherwise returns false.
- The `flush()` method flushes the file, finishing outstanding reading and writing.

Additionally, the file type supports the methods `read`, `readln`, `write`, and `writeln` for input and output as discussed in §23.5 and §23.6.

23.2 Standard files *stdout*, *stdin*, and *stderr*

The files `stdout`, `stdin`, and `stderr` are predefined and map to standard output, standard input, and standard error as implemented in a platform dependent fashion.

23.3 The *write*, *writeln*, *read*, and *readln* functions

The built-in function `write` can take an arbitrary number of arguments and writes each of the arguments out in turn to `stdout`. The built-in function `writeln` has the same semantics as `write` but outputs an *end-of-line* character after writing out the arguments. The built-in function `read` can take an arbitrary number of arguments and reads each of the arguments in turn from `stdin`. The built-in function `readln` also takes an arbitrary number of arguments, reading each argument from `stdin`. These arguments may be entered on a single line or on multiple lines. After all arguments of the `readln` call are read, an end-of-line character is expected to be read, ignoring any additional input between the last argument read and the end-of-line character.

The `read` and `readln` functions are also defined to take an arbitrary number of types as arguments. In this case, the semantics are the same except that the value returned is a tuple of the values that were read. If only one type is read, the value is not returned in a tuple, but is returned directly.

These functions are wrappers for the methods on files described next.

> *Example.* The `writeln` wrapper function allows for a simple implementation of the *Hello-World* program:
>
> ```
> writeln("Hello, World!");
> ```

23.4 User-Defined *writeThis* methods

To define the output for a given type, the user must define a method called `writeThis` on that type that takes a single argument of `Writer` type. If such a method does not exist, a default method is created.

23.5 The *write* and *writeln* method on files

The `file` type supports methods `write` and `writeln` for output. These methods are defined to take an arbitrary number of arguments. Each argument is written in turn by calling the `writeThis` method on that argument. Default `writeThis` methods are bound to any type that the user does not explicitly create one for.

A lock is used to ensure that output is serialized across multiple tasks.

23.5.1 The *write* and *writeln* method on strings

The `write` and `writeln` methods can also be called on strings to write the output to a string instead of a file.

23.5.2 Generalized *write* and *writeln*

The `Writer` class contains no arguments and serves as a base class to allow user-defined classes to be written to. If a class is defined to be a subclass of Writer, it must override the `writeIt` method that takes a `string` as an argument.

> *Example.* The following code defines a subclass of `Writer` that overrides the `writeIt` method to allow it to be written to. It also overrides the `writeThis` method to override the default way that it is written.
>
> ```
> class C: Writer {
>   var data: string;
>   def writeIt(s: string) {
>     data += s.substring(1);
>   }
>   def writeThis(x: Writer) {
>     x.write(data);
>   }
> }
>
> var c = new C();
> c.write(41, 32, 23, 14);
> writeln(c);
> ```
>
> The `C` class filters the arguments sent to it, printing out only the first letter. The output to the above is thus `4321`.

23.6 The *read* and *readln* methods on files

The `file` type supports `read` and `readln` methods. The `read` method takes an arbitrary number of arguments, reading in each argument from file. The `readln` method also takes an arbitrary number of arguments, reading in each argument from a single line or multiple lines in the file and advancing the file pointer to the next line after the last argument is read.

The `file` type also supports overloaded methods `read` and `readln` that take an arbitrary number of types as arguments. These methods read values of the specified types from the file and return them in a tuple. If only one type is read, the value is not returned in a tuple, but is returned directly.

> *Example.* The following line of code reads a value of type `int` from `stdin` and uses it to initialize variable `x` (causing `x` to have an inferred type of `int`):
>
> ```
> var x = stdin.read(int);
> ```

23.7 Default *read* and *write* methods

Default `write` methods are created for all types for which a user `write` method is not defined. They have the following semantics:

- **arrays** Outputs the elements of the array in row-major order where rows are separated by line-feeds and blank lines are used to separate other dimensions.

- **domains** Outputs the dimensions of the domain enclosed by `[` and `]`.
- **ranges** Outputs the lower bound of the range followed by `..` followed by the upper bound of the range. If the stride of the range is not one, the output is additionally followed by the word `by` followed by the stride of the range.
- **tuples** Outputs the components of the tuple in order delimited by `(` and `)`, and separated by commas.
- **classes** Outputs the values within the fields of the class prefixed by the name of the field and the character `=`. Each field is separated by a comma. The output is delimited by `{` and `}`.
- **records** Outputs the values within the fields of the class prefixed by the name of the field and the character `=`. Each field is separated by a comma. The output is delimited by `(` and `)`.

Default `read` methods are created for all types for which a user `read` method is not defined. The default `read` methods are defined to read in the output of the default `write` method.

24 Task Parallelism and Synchronization

Chapel supports both task parallelism and data parallelism, and the mixing thereof. This chapter details tasks parallelism in four parts:

- §24.1 describes the begin statement, an unstructured way to introduce concurrency into a program, and synchronization variables, an unstructured mechanism for synchronizing a program.
- §24.2 describes the cobegin and coforall statement, structured ways to introduce concurrency into a program, and the sync- and serial statement, structured ways to control and suppress parallelism.
- §24.3 describes the atomic statement, a construct to support atomic transactions.
- §24.4 describes the memory consistency model.

The term *task* is used to refer to a distinct context of execution that may be running concurrently.

24.1 Unstructured Task-Parallel Constructs

Chapel provides a simple construct, the begin statement, to spawn tasks, thus introducing concurrency into a program in an unstructured way. In addition, Chapel introduces two type qualifiers, `sync` and `single`, for synchronization of tasks.

More structured ways to achieve concurrency are discussed in §24.2. These structured ways to introduce concurrency may be easier to use in many common cases. They can be implemented using only the unstructured constructs described in this section.

24.1.1 The Begin Statement

The begin statement spawns a task to execute a statement. The begin statement is thus an unstructured way to create a new task that is executed only for its side-effects. The syntax for the begin statement is given by

begin-statement:
  begin *statement*

Control continues concurrently with the statement following the begin statement.

Example. The code

```
begin writeln("output from spawned task");
writeln("output from main task");
```

executes two `writeln` statements that output the strings to the terminal, but the ordering is purposely unspecified. There is no guarantee as to which statement will execute first. When the begin statement is executed, a new task is created that will execute the `writeln` statement within it. However, execution will continue immediately with the next statement. In §24.1.2, this same example will be synchronized so that the output from the spawned task always happens second.

The following statements may not be lexically enclosed in begin statements: break statements, continue statements, yield statements, and return statements.

24.1.2 Sync Variables

The use of and assignment to variables of `sync` type implicitly control the execution order of a task, making them well-suited to producer-consumer data sharing.

A sync variable is logically either *full* or *empty*. When it is empty, tasks that attempt to read that variable are suspended until the variable becomes full by the next assignment to it, which atomically changes the state to full. When the variable is full, a read of that variable consumes the value and atomically transitions the state to empty. If there is more than one task waiting on a sync variable, one is non-deterministically selected to use the variable and resume execution. The other tasks continue to wait for the next assignment.

If a task attempts to assign to a sync variable that is full, the task is suspended and the assignment is delayed. When the sync variable becomes empty, the task is resumed and the assignment proceeds, transitioning the state back to full. If there are multiple tasks attempting such an assignment, one is non-deterministically selected to proceed and the other assignments continue to wait until the sync variable is emptied again.

A sync variable is specified with a sync type given by the following syntax:

sync-type:
  sync *type-specifier*

If a sync variable declaration has an initialization expression, then the variable is initially full, otherwise it is initially empty.

Example. The code

```
var finishedMainOutput$: sync bool;
begin {
  finishedMainOutput$;
  writeln("output from spawned task");
}
writeln("output from main task");
finishedMainOutput$ = true;
```

modifies the example in §24.1.1. When the read of the sync variable is encountered in the spawned task, the task waits until the sync variable is assigned in the main task.

Example. Sync variables are useful for tallying data from multiple tasks as well. A sync variable of type `int` is read and then written during an update so the full-empty semantics make these updates atomic when used in a stylized way. The code

```
var count$: sync int = 0;
begin count$ += 1;
begin count$ += 1;
begin count$ += 1;
```

spawns three tasks to increment `count$`. If `count$` was not a sync variable, this code would be unsafe because between the points at which one task reads `count$` and writes `count$`, another task may increment it.

If the base type of a sync type is a class or a record, the sync semantics only apply to the class or record, not to its individual fields or methods. A record or class type may have fields of sync type to get sync semantics on individual field accesses.

If a formal argument is a sync type, the actual is passed by reference and the argument itself is a valid lvalue. The unqualified type `sync` can also be used to specify a generic formal argument. In this case, the actual must be a sync variable and it is passed by reference.

For generic formal arguments with unspecified types (§22.1.5), an actual that is `sync` is "read" before being passed to the function and the generic formal argument's type is set to the base type of the actual.

24.1.3 Single Variables

A single (assignment) variable specializes sync variables by restricting the number of times it can be assigned to no more than one during its lifetime. A use of a single variable before it is assigned causes the task's execution to suspend until the variable is assigned. Otherwise, the use proceeds as with normal variables and the task continues. After a single assignment variable is assigned, all tasks with pending uses resume in an unspecified order. A single variable is specified with a single type given by the following syntax:

single-type:
  single *type-specifier*

Example. In the code

```
class Tree {
  var isLeaf: bool;
  var left, right: Tree;
  var value: int;

  def sum():int {
    if (isLeaf) then
       return value;

    var x$: single int;
    begin x$ = left.sum();
    var y = right.sum();
    return x$+y;
  }
}
```

the single variable `x$` is assigned by an asynchronous task created with the begin statement. The task returning the sum waits on the reading of `x$` until it has been assigned.

24.1.4 Predefined Single and Sync Methods

The following methods are defined for variables of sync and single type.

```
def (sync t).readFE(): t
```

Wait for full, leave empty, and return the value of the sync variable. This method blocks until the sync variable is full. The state of the sync variable is set to empty when this method completes.

```
def (sync t).readFF(): t
def (single t).readFF(): t
```

Returns the value of the sync or single variable. This method blocks until the sync or single variable is full. The state of the sync or single variable remains full when this method completes.

```
def (sync t).readXX(): t
def (single t).readXX(): t
```

Returns the value of the sync or single variable. This method is non-blocking and the state of the sync or single variable is unchanged when this method completes.

```
def (sync t).writeEF(v: t)
def (single t).writeEF(v: t)
```

Assigns `v` to the value of the sync or single variable. This method blocks until the sync or single variable is empty. The state of the sync or single variable is set to full when this method completes.

```
def (sync t).writeFF(v: t)
```

Assigns `v` to the value of the sync variable. This method blocks until the sync variable is full. The state of the sync variable remains full when this method completes.

```
def (sync t).writeXF(v: t)
```

Assigns `v` to the value of the sync variable. This method is non-blocking and the state of the sync variable is set to full when this method completes.

```
def (sync t).reset()
```

Assigns the default value of type `t` to the value of the sync variable. This method is non-blocking and the state of the sync variable is set to empty when this method completes.

```
def (sync t).isFull: bool
def (single t).isFull: bool
```

Returns `true` if the sync or single variable is full and `false` otherwise. This method is non-blocking and the state of the sync or single variable is unchanged when this method completes.

Rationale. In general, these methods are provided such that other traditional synchronization primitives, such as semaphores and mutexes, can be constructed.

Note that the implicitly-invoked `writeEF` and `readFE`/`readFF` methods (for `sync` and `single` variables, respectively) could be considered unnecessary due to their implicit invocations, yet are provided to support programmers who wish to make the semantics of these operations more explicit. It might be desirable to have a compiler option that disables the implicit application of these methods.

Example. Given the following declarations

```
var x$: sync int;
var y$: single int;
var z: int;
```

the code

```
x$ = 5;
y$ = 6;
z = x$ + y$;
```

is equivalent to

```
x$.writeEF(5);
y$.writeEF(6);
z = x$.readFE() + y$.readFF();
```

24.2 Structured Task-Parallel Constructs

Chapel provides two constructs, the cobegin and coforall statements, to introduce concurrency in a more structured way. These constructs spawn multiple tasks but do not continue until the tasks have completed. In addition, Chapel provides two constructs, the sync- and serial statements, to suppress parallelism and insert synchronization. All four of these constructs can be implemented through judicious uses of the unstructured task-parallel constructs described in the previous section.

24.2.1 The Cobegin Statement

The cobegin statement is used to introduce concurrency within a block. The `cobegin` statement syntax is

cobegin−statement:
  cobegin *block−statement*

Each statement within the block statement is executed concurrently and is considered a separate task. Control continues when all of the tasks have finished.

The following statements may not be lexically enclosed in cobegin statements: break statements, continue statements, and return statements. Yield statement may only be lexically enclosed in cobegin statements in parallel iterators §21.4.

Example. The cobegin statement

```
cobegin {
  stmt1();
  stmt2();
  stmt3();
}
```

is equivalent to the following code that uses only begin statements and single variables to introduce concurrency and synchronize:

```
var s1$, s2$, s3$: single bool;
begin { stmt1(); s1$ = true; }
begin { stmt2(); s2$ = true; }
begin { stmt3(); s3$ = true; }
s1$; s2$; s3$;
```

Each begin statement is executed concurrently but control does not continue past the final line above until each of the single variables is written, thereby ensuring that each of the functions has finished.

24.2.2 The Coforall Loop

The coforall loop is a variant of the cobegin statement and a loop. The syntax for the coforall loop is given by

coforall−statement:
  coforall *index−var−declaration* **in** *iterator−expression* **do** *statement*
  coforall *index−var−declaration* **in** *iterator−expression block−statement*
  coforall *iterator−expression* **do** *statement*
  coforall *iterator−expression block−statement*

The semantics of the `coforall` loop are identical to a `cobegin` statement where each iteration of the `coforall` loop is equivalent to a separate statement in a `cobegin` block.

Control continues with the statement following the `coforall` loop only after all iterations have been completely evaluated.

The following statements may not be lexically enclosed in coforall statements: break statements, continue statements, and return statements. Yield statement may only be lexically enclosed in coforall statements in parallel iterators §21.4.

Example. The coforall statement

```
coforall i in iterator() {
  body();
}
```

is equivalent to the following code that uses only begin statements and sync and single variables to introduce concurrency and synchronize:

```
var runningCount$: sync int = 1;
var finished$: single bool;
for i in iterator() {
  runningCount$ += 1;
  begin {
    body();
    var tmp = runningCount$;
    runningCount$ = tmp-1;
    if tmp == 1 then finished$ = true;
  }
}
var tmp = runningCount$;
runningCount$ = tmp-1;
if tmp == 1 then finished$ = true;
finished$;
```

Each call to `body()` executes concurrently because it is in a begin statement. The sync variable `runningCount$` is used to keep track of the number of executing tasks plus one for the main task. When this variable reaches zero, the single variable `finished$` is used to signal that all of the tasks have completed. Thus control does not continue past the last line until all of the tasks have completed.

24.2.3 The Sync Statement

The sync statement acts as a join of all dynamically encountered begins from within a statement. The syntax for the sync statement is given by

```
sync-statement:
  sync statement
```

The following statements may not be lexically enclosed in sync statements: break statements, continue statements, and return statements. Yield statement may only be lexically enclosed in sync statements in parallel iterators §21.4.

Example. The sync statement can be used to wait for many dynamically spawned tasks. Given the `Tree` class defined in the example in §24.1.3 and an instance of this class called `tree`, the code

```
def concurrentUpdate(tree: Tree) {
  if requiresUpdate(tree) then
    begin update(tree);
  if !tree.isLeaf {
    concurrentUpdate(tree.left);
    concurrentUpdate(tree.right);
  }
}

sync concurrentUpdate(tree);
```

defines a function `concurrentUpdate` that recursively walks over a tree and spawns a new task to update a node if the function `requiresUpdate` evaluates to true. (Both `requiresUpdate` and `update` are omitted as irrelevant.) The call to `concurrentUpdate` is made within a `sync` statement to ensure that each of the spawned update tasks finishes before execution continues.

Example. The sync statement

```
sync {
  begin stmt1();
  begin stmt2();
}
```

is similar to the following cobegin statement

```
cobegin {
  stmt1();
  stmt2();
}
```

except that if begin statements are dynamically encountered when `stmt1()` or `stmt2()` are executed, then the former code will wait for these begin statements to complete whereas the latter code will not.

24.2.4 The Serial Statement

The `serial` statement can be used to dynamically disable parallelism. The syntax is:

serial-statement:
  serial *expression* **do** *statement*
  serial *expression block-statement*

where the expression evaluates to a bool type. Independent of that value, the *statement* is evaluated. If the expression is true, any dynamically encountered code that would result in new tasks is executed without spawning any new tasks. In effect, execution is serialized.

Example. Given the `Tree` class defined in the example in §24.1.3 and an instance of this class called `tree`, the code

```
def concurrentUpdate(tree: Tree, depth: int = 1) {
  if requiresUpdate(tree) then
    update(tree);
  if !tree.isLeaf {
    serial depth > 4 do cobegin {
      concurrentUpdate(tree.left, depth+1);
      concurrentUpdate(tree.right, depth+1);
    }
  }
}
```

defines a function `concurrentUpdate` that recursively walks over a tree using cobegin statements to update the left and right subtrees in parallel. The serial statement inhibits concurrent execution on the tree for nodes that are deeper than four levels in the tree. This constrains the number of tasks that will be used for the update.

Example. The code

```
serial true {
  begin stmt1();
  cobegin {
    stmt2();
    stmt3();
  }
  coforall i in 1..n do stmt4();
  forall i in 1..n do stmt5();
}
```

is equivalent to

```
stmt1();
{
  stmt2();
  stmt3();
}
for i in 1..n do stmt4();
for i in 1..n do stmt5();
```

because the expression evaluated to determine whether to serialize always evaluates to true.

24.3 Atomic Statements

The atomic statement creates an atomic transaction of a statement. The statement is executed with transaction semantics in that the statement executes entirely, the statement appears to have completed in a single order and serially with respect to other atomic statements, and no variable assignment is visible until the statement has completely executed.

Open issue. This definition of an atomic statement is sometimes called *strong atomicity* because the semantics are atomic to the entire program. *Weak atomicity* is defined so that an atomic statement is atomic only with respect to other atomic statements. Chapel semantics are still under design.

The syntax for the atomic statement is given by:

atomic-statement:
  atomic *statement*

Example. The following code illustrates one possible use of atomic statements:

```
var found = false;
atomic {
  if head == obj {
    found = true;
    head = obj.next;
  } else {
    var last = head;
    while last != nil {
      if last.next == obj {
        found = true;
        last.next = obj.next;
        break;
      }
      last = last.next;
    }
  }
}
```

Inside the atomic statement is a sequential implementation of removing a particular object denoted by `obj` from a singly linked list. This is an operation that is well-defined, assuming only one task is attempting it at a time. The atomic statement ensures that, for example, the value of `head` does not change after it is first in the first comparison and subsequently read to initialize `last`. The variables eventually owned by this task are `found`, `head`, `obj`, and the various `next` fields on examined objects.

The effect of an atomic statement is dynamic.

Example. If there is a method associated with a list that removes an object, that method may not be parallel safe, but could be invoked safely inside an atomic statement:

```
atomic found = head.remove(obj);
```

24.4 Memory Consistency Model

Open issue. This section is largely forthcoming.

We have been greatly helped in the design of Chapel's memory consistency model by discussions in and readings for a seminar at the University of Washington run by Dan Grossman and Luis Ceze as well as the following paper: **Jeremy Mason, William Pugh, and Sarita V. Adve. The Java memory model.** In **Proceedings of the 32nd Symposium on Principles of Programming Languages**. 2005.

The Chapel memory consistency model is defined for programs that are *data-race-free*. Programs that are *data-race-free* are sequentially consistent. Otherwise, the program is incorrect and no guarantees are made. In this design choice, Chapel differs from Java because the set of dynamic security concerns is different.

Writing and reading `sync` and `single` variables as well as executing atomic statements are the only ways in Chapel to correctly synchronize a program. It is an error to write to the same memory location or read from and write to the same memory location in two different tasks without any intervening synchronization.

Example. This has the direct consequence that one task cannot spin-wait on a variable while another task writes to that variable. The behavior of the following code is undefined:

```
var x: int;
cobegin {
  while x != 1 do ;  // spin wait
  x = 1;
}
```

While codes are more efficient in most cases if one avoids spin-waiting altogether, this code could be rewritten with defined behavior as follows:

```
var x$: sync int;
cobegin {
  while x$.readXX() != 1 do ; // spin wait
  x$.writeXF(1);
}
```

In this code, the first statement in the cobegin statement executes a loop until the variable is set to one. The second statement in the cobegin statement sets the variable to one. Neither of these statements block.

25 Data Parallelism

Chapel provides two explicit data-parallel constructs (the forall-statement and the forall-expression) and several idioms that support data parallelism implicitly (whole-array assignment, function and operator promotion, reductions, and scans).

25.1 The Forall Statement

The forall statement is a concurrent variant of the for statement described in §11.8.

25.1.1 Syntax

The syntax of the forall statement is given by

> *forall-statement*:
> **forall** *index-var-declaration* **in** *iterator-expression* **do** *statement*
> **forall** *index-var-declaration* **in** *iterator-expression block-statement*
> **forall** *iterator-expression* **do** *statement*
> **forall** *iterator-expression block-statement*
> [*index-var-declaration* **in** *iterator-expression*] *statement*
> [*iterator-expression*] *statement*

As with the for statement, the indices may be omitted if they are unnecessary and the `do` keyword may be omitted before a block statement. The bracketed form is a syntactic convenience.

25.1.2 Execution and Serializability

The forall statement evaluates the loop body once for each element in the *iterator-expression*. Each instance of the forall loop's statement may be executed concurrently with each other, but this is not guaranteed. The loop must be serializable. The definition of the iterator determines the actual concurrency based on the specification of the iterator of the loop.

This differs from the semantics of the `coforall` loop, discussed in §24.2.2, where each iteration is guaranteed to run using distinct tasks. The `coforall` loop thus has potentially higher overhead than a forall loop, but in cases where concurrency is required for correctness, it is essential.

Control continues with the statement following the forall loop only after every iteration has been completely evaluated.

The following statements may not be lexically enclosed in forall statements: break statements, continue statements, and return statements. Yield statement may only be lexically enclosed in forall statements in parallel iterators §21.4.

Example. In the code

```
forall i in 1..N do
  a(i) = b(i);
```

the user has stated that the element-wise assignments can execute concurrently. This loop may be executed serially, using a distinct task for ever iteration, or somewhere in between (using a number of tasks where each task executes a number of iterations). This loop can also be written as

```
[i in 1..N] a(i) = b(i);
```

25.1.3 Parallelism

The iterator expression determines the number of tasks that implement a forall loop as well as which iterations each task computes. For ranges, default domains, and default arrays, these values can be controlled via configuration constants (§25.7).

Additionally, the iterator expression can determine the locales on which the tasks should execute its loop iterations. For ranges, default domains, and default arrays, all tasks are executed on the current locale. Domains and arrays that are distributed across multiple locales will typically implement forall loops with multiple tasks on multiple locales.

25.1.4 Zipper Iteration

Zipper iteration has the same semantics as described in §11.8.1. With respect to parallelism, the left-most iterator expression determines the number of tasks, the iterations each task executes, and the locales on which these tasks execute.

25.1.5 Tensor Product Iteration

Tensor product iteration has the same semantics as described in §11.8.2. All iteration expressions impact parallelism as tensor product iteration is equivalent to nested forall loops. The degree of nested parallelism for ranges, default domains, and default arrays can be controlled via the configuration constant `dataParIgnoreRunningTasks` (§25.7).

25.2 The Forall Expression

The forall expression is a concurrent variant of the for expression described in §10.21.

25.2.1 Syntax

The syntax of a forall expression is given by

forall-expression:
  forall *index-var-declaration* **in** *iterator-expression* **do** *expression*
  forall *iterator-expression* **do** *expression*
  [*index-var-declaration* **in** *iterator-expression*] *expression*
  [*iterator-expression*] *expression*

As with the for expression, the indices may be omitted if they are unnecessary. The `do` keyword is always required. The bracketed form is a syntactic convenience.

25.2.2 Execution, Serializability, and Parallelism

As with the forall statement, the forall expression must be serializable. In addition, the iterator expression determines the number of tasks, the iterations each task executes, and the locales on which these tasks execute. When multiple iterator expressions are used in a zipper context, the left-most iterator determines the number of tasks, the iterations each task executes, and the locales on which these tasks execute.

The semantics are equivalent to calling a parallel iterator where the loop expression is yielded (§21.4).

Example. The code

```
writeln(+ reduce [i in 1..10] i**2);
```

applies a reduction to a forall-expression that evaluates the square of the indices in the range `1..10`.

25.2.3 Filtering Predicates in Forall Expressions

An if expression that is immediately enclosed by a forall expression does not require an else part.

Example. The following expression returns every other element starting with the first:

```
[i in 1..s.numElements] if i % 2 == 1 then s(i)
```

25.3 Whole Array Assignment

Whole array assignment is implicitly parallel. The assignment statement

```
LHS = RHS;
```

is equivalent to

```
forall (e1,e2) in (LHS,RHS) do
  e1 = e2;
```

25.4 Promotion

A function that expects one or more scalar argument but is called with one or more arrays, domains, ranges, or iterators is promoted if the element types of the arrays, the index types of the domains and/or ranges, and the yielded types of the iterators can resolve to the scalar type of the argument. The rules of when an overloaded function is promoted are discussed in §13.14.

If a promoted function returns a value, the promoted function becomes an iterator that is controlled by a loop over the iterator (or array, domain, or range) that it is promoted by. If the function does not return a value, the function is controlled by a loop over the iterator that it is promoted by, but the promotion does not become an iterator.

In addition to scalar promotion of functions, operators and casts are also promoted.

Example. Given the array

```
var A: [1..5] int = [i in 1..5] i;
```

and the function

```
def square(x: int) return x**2;
```

then the call `square(A)` results in the promotion of the `square` function over the values in the array `A`. The result is an iterator that returns the values `1`, `4`, `9`, `16`, and `25`.

Whole array operations are a form of promotion.

25.4.1 Zipper Promotion

Consider a function `f` with formal arguments `s1`, `s2`, ... that are promoted and formal arguments `a1`, `a2`, ... that are not promoted. The call

```
f(s1, s2, ..., a1, a2, ...)
```

is equivalent to

```
[(e1, e2, ...) in (s1, s2, ...)] f(e1, e2, ..., a1, a2, ...)
```

The usual constraints of zipper iteration apply to zipper promotion so the promoted actuals must have the same shape.

Example. Given a function defined as

```
def foo(i: int, j: int) {
  write(i, " ", j, " ");
}
```

and a call to this function written

```
foo(1..3, 4..6);
```

then the output is "1 4 2 5 3 6 ".

25.4.2 Tensor Product Promotion

If the function `f` were called by using square brackets instead of parentheses, the equivalent rewrite would be

```
[(e1, e2, ...) in [s1, s2, ...]] f(e1, e2, ..., a1, a2, ...)
```

There are no constraints on tensor product promotion.

Example. Given a function defined as

```
def foo(i: int, j: int) {
  write(i, " ", j, " ");
}
```

and a call to this function written

```
foo[1..3, 4..6];
```

then the output is "1 4 1 5 1 6 2 4 2 5 2 6 3 4 3 5 3 6 ".

25.5 Reductions and Scans

Chapel provides reduction and scan expressions that apply operators to aggregate expressions in stylized ways. Reduction expressions collapse the aggregate's values down to a summary value. Scan expressions compute an aggregate of results where each result value stores the result of a reduction applied to all of the elements in the aggregate up to that expression. Chapel provides a number of built-in reduction and scan operators, and also supports a mechanism for the user to define additional reductions and scans (§28).

25.5.1 Reduction Expressions

A reduction expression applies a reduction operator to an aggregate expression, collapsing the aggregate's dimensions down into a result value (typically a scalar or summary expression that is independent of the input aggregate's size). For example, a sum reduction computes the sum of all the elements in the input aggregate expression.

The syntax for a reduction expression is given by:

reduce-expression:
  reduce-scan-operator **reduce** *expression*
  class-type **reduce** *expression*

reduce-scan-operator: *one of*
  + ∗ && || & | ˆ **min max minloc maxloc**

Chapel's built-in reduction operators are defined by *reduce-scan-operator* above. In order, they are: sum, product, logical-and, logical-or, bitwise-and, bitwise-or, bitwise-exclusive-or, minimum, maximum, minimum-with-location, and maximum-with-location. The minimum reduction returns the minimum value as defined by the < operator. The maximum reduction returns the maximum value as defined by the > operator. The minimum-with-location reduction returns the lowest index position with the minimum value (as defined by the < operator). The maximum-with-location reduction returns the lowest index position with the maximum value (as defined by the > operator).

The expression on the right-hand side of the `reduce` keyword can be of any type that can be iterated over and to which the reduction operator can be applied. For example, the bitwise-and operator can be applied to arrays of boolean or integral types to compute the bitwise-and of all the values in the array.

The minimum-with-location and maximum-with-location reductions take a 2-tuple of arguments where the first tuple element is the collection of values for which the minimum/maximum value is to be computed. The second tuple element is a collection of indices with the same size and shape that provides names for the locations of the values in the first argument. The reduction returns a tuple containing the minimum/maximum value in the first position and the location of the value in the second position.

Example. The first line below computes the smallest element in an array `A` as well as its index, storing the results in `minA` and `minALoc`, respectively. It then computes the largest element in a forall expression making calls to a function `foo()`, storing the value and its number in `maxVal` and `maxValNum`.

```
var (minA, minALoc) = minloc reduce (A, A.domain);
var (maxVal, maxValNum) = maxloc reduce ([i in 1..n] foo(i), 1..n);
```

User-defined reductions are specified by preceding the keyword `reduce` by the class type that implements the reduction interface as described in §28.

25.5.2 Scan Expressions

A scan expression applies a scan operator to an aggregate expression, resulting in an aggregate expression of the same size and shape. The output values represent the result of the operator applied to all elements up to and including the corresponding element in the input.

The syntax for a scan expression is given by:

```
scan-expression:
  reduce-scan-operator scan expression
  class-type scan expression
```

The built-in scans are defined in *reduce-scan-operator*. These are identical to the built-in reductions and are described in §25.5.1.

The expression on the right-hand side of the scan can be of any type that can be iterated over and to which the operator can be applied.

User-defined scans are specified by preceding the keyword `scan` by the class type that implements the scan interface as described in §28.

Example. Given an array

```
var A: [1..3] int = 1;
```

that is initialized such that each element contains one, then the code

```
writeln(+ scan A);
```

outputs the results of scanning the array with the sum operator. The output is

```
1 2 3
```

25.6 Data Parallelism and Evaluation Order

Temporary arrays are never inserted by the Chapel compiler. The semantics of whole array assignment, promotion, etc., are thus different than in array programming languages.

Example. If `A` is an array declared over the indices `1..5`, then the following codes are not equivalent:

```
A[2..4] = A[1..3] + A[3..5];
```

and

```
var T = A[1..3] + A[3..5];
A[2..4] = T;
```

This follows because, in the former code, some of the new values that are assigned to `A` may be read to compute the sum depending on the number of tasks used to implement the data parallel statement.

25.7 Knobs for Default Data Parallelism

The following configuration constants are provided to control the degree of data parallelism over ranges, default domains, and default arrays:

Config Const	**Type**	**Default**
`dataParTasksPerLocale`	`int`	Number of cores per locale
`dataParIgnoreRunningTasks`	`bool`	`true`
`dataParMinGranularity`	`int`	`1`

The configuration constant `dataParTasksPerLocale` specifies the number of tasks to use when executing a forall loop over a range, default domain, or default array. The actual number of tasks may be fewer depending on the two other configuration constants. A value of zero results in using the default value.

The configuration constant `dataParIgnoreRunningTasks`, when true, has no effect on the number of tasks to use to execute the forall loop. When false, the number of tasks per locale is decreased by the number of tasks that are already running on the locale.

The configuration constant `dataParMinGranularity` specifies the minimum number of iterations per task created. The number of tasks is decreased so that the number of iterations per task is never less than the specified value.

For distributed domains and arrays that have these same knobs (*e.g.*, using the Block and Cyclic distributions), these same global configuration constants are used to specify their default behavior within each locale.

26 Locales

Chapel provides high-level abstractions that allow programmers to exploit locality by controlling the affinity of both data and tasks to abstract units of processing and storage capabilities called *locales*. The *on-statement* allows for the migration of tasks to *remote* locales.

Throughout this section, the term *local* will be used to describe the locale on which a task is running, the data located on this locale, and any tasks running on this locale. The term *remote* will be used to describe another locale, the data on another locale, and the tasks running on another locale.

26.1 Locales

A *locale* is a portion of the target parallel architecture that has processing and storage capabilities. Chapel implementations should typically define locales for a target architecture such that tasks running within a locale have roughly uniform access to values stored in the locale's local memory and longer latencies for accessing the memories of other locales. As an example, a cluster of multicore nodes or SMPs would typically define each node to be a locale. In contrast a pure shared memory machine would be defined as a single locale.

26.1.1 Locale Types

The identifier `locale` is a primitive type that abstracts a locale as described above. Both data and tasks can be associated with a value of locale type. The only operators defined over locales are the equality and inequality comparison operators.

26.1.2 Locale Methods

The locale type supports the following methods:

```
def locale.id: int;
```

Returns a unique integer for each locale, from 0 to the number of locales less one.

```
def locale.numCores: int;
```

Returns the number of processor cores available on a given locale.

```
use Memory;
def locale.physicalMemory(unit: MemUnits=MemUnits.Bytes, type retType=int(64)): retType;
```

Returns the amount of physical memory available on a given locale in terms of the specified memory units (Bytes, KB, MB, or GB) using a value of the specified return type.

26.1.3 The Predefined Locales Array

Chapel provides a predefined environment that stores information about the locales used during program execution. This *execution environment* contains definitions for the array of locales on which the program is executing (`Locales`), a domain for that array (`LocaleSpace`), and the number of locales (`numLocales`).

```
config const numLocales: int;
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

When a Chapel program starts, a single task executes `main` on `Locales(0)`.

Note that the Locales array is typically defined such that distinct elements refer to distinct resources on the target parallel architecture. In particular, the Locales array itself should not be used in an oversubscribed manner in which a single processor resource is represented by multiple locale values (except during development). Oversubscription should instead be handled by creating an aggregate of locale values and referring to it in place of the Locales array.

> *Rationale*. This design choice encourages clarity in the program's source text and enables more opportunities for optimization.
>
> For development purposes, oversubscription is still very useful and this should be supported by Chapel implementations to allow development on smaller machines.

> *Example*. The code
>
> ```
> const MyLocales: [loc in 0..numLocales*4] locale = Locales(loc%numLocales);
> on MyLocales(i) ...
> ```
>
> defines a new array `MyLocales` that is four times the size of the `Locales` array. Each locale is added to the `MyLocales` array four times in a round-robin fashion.

26.1.4 The *here* Locale

A predefined constant locale `here` can be used anywhere in a Chapel program. It refers to the locale that the current task is running on.

> *Example*. The code
>
> ```
> on Locales(1) {
>   writeln(here.id);
> }
> ```
>
> results in the output `1` because the `writeln` statement is executed on locale 1.

The identifier `here` is not a keyword and can be overridden.

26.1.5 Querying the Locale of an Expression

The locale associated with an expression (where the expression is stored) is queried using the following syntax:

```
locale-access-expression:
  expression . locale
```

When the expression is a class, the access returns the locale on which the class object exists rather than the reference to the class. If the expression is a value, it is considered local. The implementation may warn about this behavior. If the expression is a locale, it is returned directly.

Example. Given a class C and a record R, the code

```
on Locales(1) {
  var x: int;
  var c: C;
  var r: R;
  on Locales(2) {
    on Locales(3) {
      c = new C();
      r = new R();
    }
    writeln(x.locale);
    writeln(c.locale);
    writeln(r.locale);
  }
}
```

results in the output

```
1
3
1
```

The variable `x` is declared and exists on `Locales(0)`. The variable `c` is a class reference. The reference exists on `Locales(1)` but the object itself exists on `Locales(3)`. The locale access returns the locale where the object exists. Lastly, the variable `r` is a record and has value semantics. It exists on `Locales(1)` even though it is assigned a value on a remote locale.

Global (non-distributed) constants are replicated across all locales.

Example. For example, the following code:

```
const c = 10;
for loc in Locales do on loc do
    writeln(c.locale);
```

outputs

```
LOCALE0
LOCALE1
LOCALE2
LOCALE3
LOCALE4
```

when running on 5 locales.

26.2 The On Statement

The on statement controls on which locale a block of code should be executed or data should be placed. The syntax of the on statement is given by

> *on−statement*:
> **on** *expression* **do** *statement*
> **on** *expression block−statement*

The locale of the expression is automatically queried as described in §26.1.5. Execution of the statement occurs on this specified locale and then continues after the `on-statement`.

Return statements may not be lexically enclosed in on statements. Yield statement may only be lexically enclosed in on statements in parallel iterators §21.4.

26.2.1 Remote Variable Declarations

By default, when new variables and data objects are created, they are created in the locale where the task is running. Variables can be defined within an *on−statement* to define them on a particular locale such that the scope of the variables is outside the *on−statement*. This is accomplished using a similar syntax but omitting the `do` keyword and braces. The syntax is given by:

> *remote−variable−declaration−statement*:
> **on** *expression variable−declaration−statement*

27 Domain Maps

Domain maps specify the implementation of domains and, in turn, arrays by defining the mapping from indices in domains to memory locations within or across locales. The term *layout* is used to describe a domain map that describes domains and arrays that exist on a single locale. The term *distribution* is used to describe a domain map that describes domains and arrays that are partitioned across multiple locales.

The domain map abstraction is not only used to define this mapping, but rather is used to define the implementation of domains and arrays including their accessors and iterators.

27.1 Domain Map Types

Domain map types are defined by the type of the implementing domain class, but are distinct from the class type. Typically, the domain map class type is only used on its own in defining the domain map itself. Defining a domain map is discussed in §29.

Specifying a domain map type involves specifying a domain map class type and wrapping it by the domain map specifier `dmap`.

Example. The code

```
use BlockDist;
var MyBlockDist: dmap(Block(rank=2));
```

creates a uninitialized two-dimensional Block distribution called `MyBlockDist` that can be used to distribute 2-dimensional arithmetic domains. The Block distribution is described in more detail in §31.1.

27.2 Domain Map Values

Constructing a domain map value involves calling the constructor of a domain map class and defining a new domain map type `dmap`.

Example. The code

```
use BlockDist;
var MyBlockDist: dmap(Block(rank=2)) = new dmap(new Block([1..n,1..n]));
```

creates an initialized two-dimensional Block distribution with a bounding box of `[1..n, 1..n]` over all of the locales. The Block distribution is described in more detail in §31.1.

27.3 Mapped Domains and Arrays

A domain for which a domain map is specified is referred to as a *mapped domain*.

The syntax to create a mapped domain type is the same as the syntax to create a mapped domain value:

```
mapped-domain-type:
  domain-type dmapped domain-map-expression

mapped-domain-expression:
  domain-expression dmapped domain-map-expression

domain-map-expression:
  expression
```

Example. The code

```
use BlockDist;
var MyBlockDist = new dmap(new Block([1..n,1..n]));
var Dom: domain(2) dmapped MyBlockDist = [1..n, 1..n];
```

defines a new Block-distributed domain, mapped via `MyBlockDist`.

When defining a new domain map inline with the `dmapped` keyword, a syntactic sugar is supported in which the "new dmap(new" characters (along with the closing parenthesis) may be omitted.

Example. The code

```
use BlockDist;
var D = [1..n, 1..n] dmapped new dmap(new Block([1..n,1..n]));
```

is equivalent to

```
use BlockDist;
var D = [1..n, 1..n] dmapped Block([1..n,1..n]);
```

27.4 Default Mapped Domains and Arrays

If a domain is not mapped via the `dmapped` keyword, it is implicitly mapped by a default layout to the locale on which it is declared.

28 User-Defined Reductions and Scans

User-defined reductions and scans are supported via class definitions where the class implements a structural interface. The definition of this structural interface is forthcoming. The following paper sketched out such an interface:

> S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. **Global-view abstractions for user-defined reductions and scans**. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

29 User-Defined Domain Maps

This chapter is forthcoming.

30 Standard Modules

Standard modules provide standard library support and are available to every Chapel program. The functions and other definitions of automatic modules are always available to a Chapel program. Optional modules can be accessed via use statements (§12.4).

The automatic modules are as follows:

`Math`	Math routines
`Base`	Basic routines
`Types`	Routines related to primitive types

The optional modules include:

`BitOps`	Bit manipulation routines
`Functions`	Common higher-order functions
`Norm`	Routines for computing vector and matrix norms
`Random`	Random number generation routines
`Search`	Generic searching routines
`Sort`	Generic sorting routines
`Time`	Types and routines related to time

There is an expectation that each of these modules will be extended and that more standard modules will be defined over time.

30.1 Automatic Modules

Automatic modules are used by a Chapel program automatically. There is currently no way to avoid their use by a program, although we anticipate adding such a capability in the future.

30.1.1 Math

The module `Math` defines routines for mathematical computations. This module is used by default; there is no need to explicitly use this module. The Math module defines routines that are derived from and implemented via the standard C routines defined in `math.h`.

```
def abs(i: int(?w)): int(w)
def abs(i: uint(?w)): uint(w)
def abs(x: real): real
def abs(x: real(32)): real(32)
def abs(x: complex): real
```

Returns the absolute value of the argument.

```
def acos(x: real): real
def acos(x: real(32)): real(32)
```

Returns the arc cosine of the argument. It is an error if `x` is less than -1 or greater than 1.

```
def acosh(x: real): real
def acosh(x: real(32)): real(32)
```

Returns the inverse hyperbolic cosine of the argument. It is an error if `x` is less than 1.

```
def asin(x: real): real
def asin(x: real(32)): real(32)
```

Returns the arc sine of the argument. It is an error if `x` is less than -1 or greater than 1.

```
def asinh(x: real): real
def asinh(x: real(32)): real(32)
```

Returns the inverse hyperbolic sine of the argument.

```
def atan(x: real): real
def atan(x: real(32)): real(32)
```

Returns the arc tangent of the argument.

```
def atan2(y: real, x: real): real
def atan2(y: real(32), x: real(32)): real(32)
```

Returns the arc tangent of the two arguments. This is equivalent to the arc tangent of `y / x` except that the signs of `y` and `x` are used to determine the quadrant of the result.

```
def atanh(x: real): real
def atanh(x: real(32)): real(32)
```

Returns the inverse hyperbolic tangent of the argument. It is an error if `x` is less than -1 or greater than 1.

```
def cbrt(x: real): real
def cbrt(x: real(32)): real(32)
```

Returns the cube root of the argument.

```
def ceil(x: real): real
def ceil(x: real(32)): real(32)
```

Returns the value of the argument rounded up to the nearest integer.

```
def conjg(a: complex(?w)): complex(w)
```

Returns the conjugate of `a`.

```
def cos(x: real): real
def cos(x: real(32)): real(32)
```

Returns the cosine of the argument.

```
def cosh(x: real): real
def cosh(x: real(32)): real(32)
```

Returns the hyperbolic cosine of the argument.

```
def erf(x: real): real
def erf(x: real(32)): real(32)
```

Returns the error function of the argument defined as

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

for the argument x.

```
def erfc(x: real): real
def erfc(x: real(32)): real(32)
```

Returns the complementary error function of the argument. This is equivalent to `1.0 - erf(x)`.

```
def exp(x: real): real
def exp(x: real(32)): real(32)
```

Returns the value of e raised to the power of the argument.

```
def exp2(x: real): real
def exp2(x: real(32)): real(32)
```

Returns the value of 2 raised to the power of the argument.

```
def expm1(x: real): real
def expm1(x: real(32)): real(32)
```

Returns one less than the value of e raised to the power of the argument.

```
def floor(x: real): real
def floor(x: real(32)): real(32)
```

Returns the value of the argument rounded down to the nearest integer.

```
def lgamma(x: real): real
def lgamma(x: real(32)): real(32)
```

Returns the natural logarithm of the absolute value of the gamma function of the argument.

```
def log(x: real): real
def log(x: real(32)): real(32)
```

Returns the natural logarithm of the argument. It is an error if the argument is less than or equal to zero.

```
def log10(x: real): real
def log10(x: real(32)): real(32)
```

Returns the base 10 logarithm of the argument. It is an error if the argument is less than or equal to zero.

```
def log1p(x: real): real
def log1p(x: real(32)): real(32)
```

Returns the natural logarithm of `x+1`. It is an error if `x` is less than or equal to -1.

```
def log2(i: int(?w)): int(w)
def log2(i: uint(?w)): uint(w)
def log2(x: real): real
def log2(x: real(32)): real(32)
```

Returns the base 2 logarithm of the argument. It is an error if the argument is less than or equal to zero.

```
def nearbyint(x: real): real
def nearbyint(x: real(32)): real(32)
```

Returns the rounded integral value of the argument determined by the current rounding direction.

```
def rint(x: real): real
def rint(x: real(32)): real(32)
```

Returns the rounded integral value of the argument determined by the current rounding direction.

```
def round(x: real): real
def round(x: real(32)): real(32)
```

Returns the rounded integral value of the argument. Cases halfway between two integral values are rounded towards zero.

```
def sin(x: real): real
def sin(x: real(32)): real(32)
```

Returns the sine of the argument.

```
def sinh(x: real): real
def sinh(x: real(32)): real(32)
```

Returns the hyperbolic sine of the argument.

```
def sqrt(x: real): real
def sqrt(x: real(32)): real(32)
```

Returns the square root of the argument. It is an error if the argument is less than zero.

```
def tan(x: real): real
def tan(x: real(32)): real(32)
```

Returns the tangent of the argument.

```
def tanh(x: real): real
def tanh(x: real(32)): real(32)
```

Returns the hyperbolic tangent of the argument.

```
def tgamma(x: real): real
def tgamma(x: real(32)): real(32)
```

Returns the gamma function of the argument defined as

$$\int_0^\infty t^{x-1}e^{-t}dt$$

for the argument x.

```
def trunc(x: real): real
def trunc(x: real(32)): real(32)
```

Returns the nearest integral value to the argument that is not larger than the argument in absolute value.

30.1.2 Base

def `ascii(s:` **string**`):` **int**

Returns the ASCII code number of the first letter in the argument `s`.

def `assert(test:` **bool**`) {`

Exits the program if `test` is false and prints to standard error the location in the Chapel code of the call to `assert`. If `test` is true, no action is taken.

def `assert(test:` **bool**`, args ...?numArgs) {`

Exits the program if `test` is false and prints to standard error the location in the Chapel code of the call to `assert` as well as the rest of the arguments to the call. If `test` is true, no action is taken.

def complex`.re:` **real**

Returns the real component of the complex number.

def complex`.im:` **real**

Returns the imaginary component of the complex number.

def complex`.=re(f:` **real**`)`

Sets the real component of the complex number to `f`.

def complex`.=im(f:` **real**`)`

Sets the imaginary component of the complex number to `f`.

def `exit(status:` **int**`)`

Exits the program with code `status`.

def `halt()`

Exits the program and prints to standard error the location in the Chapel code of the call to `halt`.

def `halt(args ...?numArgs)`

Exits the program and prints to standard error the location in the Chapel code of the call to `halt` as well as the rest of the arguments to the call.

def string`.length:` **int**

Returns the number of characters in the base expression of type string.

def `max(x, y...?k)`

Returns the maximum of the arguments when compared using the “greater-than” operator. The return type is inferred from the types of the arguments as allowed by implicit coercions.

def `min(x, y...?k)`

Returns the minimum of the arguments when compared using the “less-than” operator. The return type is inferred from the types of the arguments as allowed by implicit coercions.

def string`.substring(x):` **string**

Returns a value of string type that is a substring of the base expression. If `x` is i, a value of type `int`, then the result is the ith character. If `x` is a range, the result is the substring where the characters in the substring are given by the values in the range.

def `typeToString(`**type** `t)` **param** `:` **string**

Returns a string parameter that represents the name of the type `t`.

30.1.3 Types

```
def numBits(type t) param : int
```

Returns the number of bits used to store the values of type `t`. This is implemented for all numeric types and fixed-width `bool` types. It is not implemented for default-width `bool`.

```
def numBytes(type t) param : int
```

Returns the number of bytes used to store the values of type `t`. This is implemented for all numeric types and fixed-width `bool` types. It is not implemented for default-width `bool`.

```
def max(type t): t
```

Returns the maximum value that can be stored in type `t`. This is implemented for all numeric types.

```
def min(type t): t
```

Returns the minimum value that can be stored in type `t`. This is implemented for all numeric types.

30.2 Optional Modules

Optional modules can be used by a Chapel program via the `use` keyword (§12.4).

30.2.1 BitOps

The module `BitOps` defines routines that manipulate the bits of values of integral types.

```
def bitPop(i: integral): int
```

Returns the number of bits set to one in the integral argument `i`.

```
def bitMatMultOr(i: uint(64), j: uint(64)): uint(64)
```

Returns the bitwise matrix multiplication of `i` and `j` where the values of `uint(64)` type are treated as 8×8 bit matrices and the combinator function is bitwise or.

```
def bitRotLeft(i: integral, shift: integral): i.type
```

Returns the value of the integral argument `i` after rotating the bits to the left `shift` number of times.

```
def bitRotRight(i: integral, shift: integral): i.type
```

Returns the value of the integral argument `i` after rotating the bits to the right `shift` number of times.

30.2.2 Norm

The module `Norm` supports the computation of standard vector and matrix norms on Chapel arrays. The current interface is minimal and should be expected to grow and evolve over time.

```
enum normType {norm1, norm2, normInf, normFrob};
```

An enumerated type indicating the different types of norms supported by this module: 1-norm, 2-norm, infinity norm and Frobenius norm, respectively.

```
def norm(x: [], p: normType) where x.rank == 1 || x.rank == 2
```

Compute the norm indicated by `p` on the 1D or 2D array `x`.

```
def norm(x: [])
```

Compute the default norm on array `x`. For a 1D array this is the 2-norm, for a 2D array, this is the Frobenius norm.

30.2.3 Random

The module `Random` supports the generation of pseudo-random values and streams of values. The current interface is minimal and should be expected to grow and evolve over time. In particular, we expect to support other pseudo-random number generation algorithms, more random value types (*e.g.*, int), and both serial and parallel iterators over the RandomStream class.

```
class RandomStream
def RandomStream(seed: int(64), param parSafe: bool = true)
def RandomStream(seedGenerator: SeedGenerator = SeedGenerator.currentTime,
                 param parSafe: bool = true)
```

Implements a pseudo-random stream of values based on a seed value. The current implementation generates the values using a linear congruential generator. In future versions of this module, the RandomStream class will offer a wider variety of algorithms for generating pseudo-random values.

To construct a RandomStream class, the seed may be explicitly passed. It must be an odd integer between 1 and $2^{46} - 1$. Alternatively, the RandomStream class can be constructed by passing a value of the enumerated type SeedGenerator to choose an algorithm to use to set the seed. If neither a seed nor a SeedGenerator value is passed to the RandomStream class, the seed will be initialized based on the current time in microseconds (rounded via modular arithmetic to the nearest odd integer between 1 and $2^{46} - 1$.

The parSafe parameter defaults to true and allows for safe use of this class by concurrent tasks. This can be overridden when calling methods to make them safe when called by concurrent tasks. This mechanism allows for lower overhead calls when there is no threat of concurrent calls, but correct calls when there is.

```
enum SeedGenerator { currentTime };
```

Values of this enumerated type may be used to choose a method for initializing the seed in the RandomStream class. The only value supported at present is `currentTime` which can be used to initialize the seed based on the current time in microseconds (rounded via modular arithmetic to the nearest odd integer between 1 and $2^{46} - 1$.

```
def RandomStream.fillRandom(x:[?D], param parSafe = this.parSafe)
```

Fill the argument array, `x`, with the next |`D`| values of the pseudo-random stream in row-major order. The array must be an array of real(64), imag(64), or complex(128) elements. For complex arrays, each complex element is initialized with two values from the stream of random numbers.

```
def RandomStream.skipToNth(in n: integral, param parSafe = this.parSafe)
```

Skips ahead or back to the `n`-th value in the random stream. The value of n is assumed to be positive, such that `n` == 1 represents the initial value in the stream.

```
def RandomStream.getNext(param parSafe = this.parSafe): real
```

Returns the next value in the random stream as a real.

```
def RandomStream.getNth(n: integral, param parSafe = this.parSafe): real
```

Returns the `n`-th value in the random stream as a real. Equivalent to calling `skipToNth(n)` followed by `getNext()`.

```
def fillRandom(x:[], initseed: int(64))
```

A routine provided for convenience to support the functionality of the fillRandom method (above) without explicitly constructing an instance of the `RandomStream` class. This is useful for filling a single array or multiple arrays which require no coherence between them. The `initseed` parameter corresponds to the `seed` member of the `RandomStream` class. If unspecified, the default for the class will be used.

30.2.4 Search

The `Search` module is designed to support standard search routines. The current interface is minimal and should be expected to grow and evolve over time.

```
def LinearSearch(Data: [?Dom], val): (bool, index(Dom))
```

Searches through the pre-sorted array `Data` looking for the value `val` using a sequential linear search. Returns a tuple indicating (1) whether or not the value was found and (2) the location of the value if it was found, or the location where the value should have been if it was not found.

```
def BinarySearch(Data: [?Dom], val, in lo = Dom.low, in hi = Dom.high)
```

Searches through the pre-sorted array `Data` looking for the value `val` using a sequential binary search. If provided, only the indices `lo` through `hi` will be considered, otherwise the whole array will be searched. Returns a tuple indicating (1) whether or not the value was found and (2) the location of the value if it was found, or the location where the value should have been if it was not found.

30.2.5 Sort

The `Sort` module is designed to support standard sorting routines. The current interface is minimal and should be expected to grow and evolve over time.

```
def InsertionSort(Data: [?Dom]) where Dom.rank == 1
```

Sorts the 1D array `Data` in-place using a sequential insertion sort algorithm.

```
def QuickSort(Data: [?Dom]) where Dom.rank == 1
```

Sorts the 1D array `Data` in-place using a sequential implementation of the QuickSort algorithm.

30.2.6 Time

The module `Time` defines routines that query the system time and a record `Timer` that is useful for timing portions of code.

```
record Timer
```

A timer is used to time portions of code. Its semantics are similar to a stopwatch.

```
enum TimeUnits { microseconds, milliseconds, seconds, minutes, hours };
```

The enumeration TimeUnits defines units of time. These units can be supplied to routines in this module to specify the desired time units.

```
enum Day { sunday=0, monday, tuesday, wednesday, thursday, friday, saturday };
```

The enumeration Day defines the days of the week, with Sunday defined to be 0.

```
def getCurrentDate(): (int, int, int)
```

Returns the year, month, and day of the month as integers. The year is the year since 0. The month is in the range 1 to 12. The day is in the range 1 to 31.

```
def getCurrentDayOfWeek(): Day
```

Returns the current day of the week.

```
def getCurrentTime(unit: TimeUnits = TimeUnits.seconds): real
```

Returns the elapsed time since midnight in the units specified.

```
def Timer.clear()
```

Clears the elapsed time stored in the Timer.

```
def Timer.elapsed(unit: TimeUnits = TimeUnits.seconds): real
```

Returns the cumulative elapsed time, in the units specified, between calls to `start` and `stop`. If the timer is running, the elapsed time since the last call to `start` is added to the return value.

```
def Timer.start()
```

Starts the timer. It is an error to start a timer that is already running.

```
def Timer.stop()
```

Stops the timer. It is an error to stop a timer that is not running.

```
def sleep(t: uint)
```

Delays a task for `t` seconds.

31 Standard Distributions

The following table lists distributions standard to the Chapel language:

Distribution	**Module**	**Supported Domain Types**
`Block`	`BlockDist`	Arithmetic
`Cyclic`	`CyclicDist`	Arithmetic

Rationale. Why supply any standard distributions? A main design goal of Chapel requires that the standard distributions be defined using the same mechanisms available to Chapel programmers wishing to define their own distributions or layouts (§29). That way there shouldn't be a necessary performance cost associated with user-defined domain maps. Nevertheless, distributions are an integral part of the Chapel language which would feel incomplete without a good set of standard distributions. It is hoped that many distributions will begin as user-defined domain maps and later become part of the standard set of distributions.

31.1 The Standard Block Distribution

The standard Block distribution, defined in the module `BlockDist`, maps indices to locales by partitioning the indices into blocks according to a *bounding box* argument. It is parameterized by the rank and index type of the domains it supports. Thus domains of different ranks or different index types must be distributed with different Block distributions.

For Block distributions of rank d, given a bounding box

`[`l_1`..`h_1`, ..., `l_d`..`h_d`]`

and an array of target locales defined over the domain

`[0..`n_1`-1, ..., 0..`$n_d - 1$`]`

then a Block distribution maps an index i to a locale by computing the *kth* component of an index j into the array of target locales from the *kth* component of i using the following formula:

$$j_k = \begin{cases} 0 & \text{if } i_k < l_k \\ \left\lfloor \dfrac{n_k(i_k - l_k)}{h_k - l_k + 1} \right\rfloor & \text{if } i_k \geq l_k \text{ and } i_k \leq h_k \\ n_k - 1 & \text{if } i_k > h_k \end{cases}$$

The Block class constructor is defined as follows:

```
def Block(boundingBox: domain,
          targetLocales: [] locale = Locales,
          dataParTasksPerLocale = value in global config const of the same name,
          dataParIgnoreRunningTasks = value in global config const of the same name,
          dataParMinGranularity = value in global config const of the same name,
          param rank = boundingBox.rank,
          type idxType = boundingBox.dim(1).eltType)
```

The argument `boundingBox` is a non-distributed domain defining a bounding box used to partition the space of all indices across an array of target locales.

The argument `targetLocales` is a non-distributed array containing the target locales to which this distribution maps indices and data. The rank of `targetLocales` must match the rank of the distribution, or be one. If the rank of `targetLocales` is one, a greedy heuristic is used to reshape the array of target locales so that it matches the rank of the distribution and each dimension contains an approximately equal number of indices.

The arguments `dataParTasksPerLocale`, `dataParIgnoreRunningTasks`, and `dataParMinGranularity` set the knobs that are used to control intra-locale data parallelism for Block-distributed domains and arrays in the same way that the global configuration constants of these names control data parallelism for ranges and default-distributed domains and arrays §25.7.

The `rank` and `idxType` arguments are inferred from the `boundingBox` argument unless explicitly set.

> *Example*. The following code declares a Block distribution with a bounding box equal to the domain `Space` and declares an array, `A`, over a domain declared over this distribution. The computation in the `forall` loop sets each array element to the ID of the locale to which it is mapped.
>
> ```
> use BlockDist;
>
> const Space = [1..8, 1..8];
> const D: domain(2) dmapped Block(boundingBox=Space) = Space;
> var A: [D] int;
>
> forall a in A do
>   a = a.locale.id;
>
> writeln(A);
> ```
>
> When run on 6 locales, the output is:
>
> ```
> 0 0 0 0 1 1 1 1
> 0 0 0 0 1 1 1 1
> 0 0 0 0 1 1 1 1
> 2 2 2 2 3 3 3 3
> 2 2 2 2 3 3 3 3
> 2 2 2 2 3 3 3 3
> 4 4 4 4 5 5 5 5
> 4 4 4 4 5 5 5 5
> ```

31.2 The Standard Cyclic Distribution

The standard Cyclic distribution, defined in the module `CyclicDist`, maps indices to locales in a round-robin pattern according to a *start index* argument. It is parameterized by the rank and index type of the domains it supports. Thus domains of different ranks or different index types must be distributed with different Cyclic distributions.

For cyclic distributions of rank d, given a start index

$(s_1, \ldots, s_d)$

and an array of target locales defined over the domain

```
[0..n1-1, ..., 0..nd-1]
```

then a Cyclic distribution maps an index i to a locale by computing the kth component of an index j into the array of target locales from the kth component of i using the following formula:

$$j_k = i_k - s_k \pmod{n_k}$$

The Cyclic class constructor is defined as follows:

```
def Cyclic(startIdx,
           targetLocales: [] locale = Locales,
           dataParTasksPerLocale = value in global config const of the same name,
           dataParIgnoreRunningTasks = value in global config const of the same name,
           dataParMinGranularity = value in global config const of the same name,
           param rank: int = rank inferred from startIdx,
           type idxType = index type inferred from startIdx)
```

The argument `startIdx` is a tuple of integers defining an index that will be distributed to the first locale in `targetLocales`. For a single dimensional distribution `startIdx` can be an integer or a tuple with a single element.

The argument `targetLocales` is a non-distributed array containing the target locales to which this distribution maps indices and data. The rank of `targetLocales` must match the rank of the distribution, or be one. If the rank of `targetLocales` is one, a greedy heuristic is used to reshape the array of target locales so that it matches the rank of the distribution and each dimension contains an approximately equal number of indices.

The arguments `dataParTasksPerLocale`, `dataParIgnoreRunningTasks`, and `dataParMinGranularity` set the knobs that are used to control intra-locale data parallelism for Cyclic-distributed domains and arrays in the same way that the global configuration constants of these names control data parallelism for ranges and default-distributed domains and arrays §25.7.

The `rank` and `idxType` arguments are inferred from the `startIdx` argument unless explicitly set.

Example. The following code declares a Cyclic distribution with a start index of `(1,1)` and declares an array, `A`, over a domain declared over this distribution. The computation in the `forall` loop sets each array element to the ID of the locale to which it is mapped.

```
use CyclicDist;

const Space = [1..8, 1..8];
const D: domain(2) dmapped Cyclic(startIdx=Space.low) = Space;
var A: [D] int;

forall a in A do
  a = a.locale.id;

writeln(A);
```

When run on 6 locales, the output is:

```
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
4 5 4 5 4 5 4 5
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
4 5 4 5 4 5 4 5
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
```

32 Standard Layouts

This chapter is forthcoming.

A Collected Lexical and Syntax Productions

This appendix collects the syntax productions listed throughout the specification. There are no new syntax productions in this appendix. The productions are listed both alphabetically and in depth-first order for convenience.

A.1 Alphabetical Lexical Productions

binary−digit: *one of*
  0 1

binary−digits:
  binary−digit
  binary−digit binary−digits

bool−literal: *one of*
  true false

digit: *one of*
  0 1 2 3 4 5 6 7 8 9

digits:
  digit
  digit digits

double−quote−delimited−characters:
  string−character double−quote−delimited−characters$_{opt}$
  ' *double−quote−delimited−characters*$_{opt}$

exponent−part:
  e *sign*$_{opt}$ *digits*
  E *sign*$_{opt}$ *digits*

hexadecimal−digit: *one of*
  0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

hexadecimal−digits:
  hexadecimal−digit
  hexadecimal−digit hexadecimal−digits

hexadecimal−escape−character:
  \x *hexadecimal−digits*

identifier:
  letter legal−identifier−chars$_{opt}$
  _ *legal−identifier−chars*

imaginary−literal:
  real−literal **i**
  integer−literal **i**

integer−literal:
  digits
  0x *hexadecimal−digits*
  0X *hexadecimal−digits*
  0b *binary−digits*
  0B *binary−digits*

legal−identifier−char:
  letter
  digit
  $

legal−identifier−chars:
  legal−identifier−char legal−identifier−chars$_{opt}$

letter: *one of*
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

real−literal:
  digits$_{opt}$. *digits exponent−part$_{opt}$*
  digits exponent−part

sign: *one of*
  + −

simple−escape−character: *one of*
  \’ \” \? \a \b \f \n \r \t \v

single−quote−delimited−characters:
  string−character single−quote−delimited−characters$_{opt}$
  ” *single−quote−delimited−characters$_{opt}$*

string−character:
  any character except the double quote, single quote, or new line
  simple−escape−character
  hexadecimal−escape−character

string−literal:
  ” *double−quote−delimited−characters$_{opt}$* ”
  ’ *single−quote−delimited−characters$_{opt}$* ’

A.2 Alphabetical Syntax Productions

argument−list:
  (*formals$_{opt}$*)

array−alias−declaration:
  identifier reindexing−expression$_{opt}$ => *array−expression* ;

array−expression:
  expression

array−type:
  [*domain−expression*] *type−specifier*

assignment−operator: *one of*
  = += −= ∗= /= %= ∗∗= &= |= ˆ= &&= ||= <<= >>=

assignment−statement:
  lvalue−expression assignment−operator expression

associative−domain−type:
  domain (*associative−index−type*)

associative−index−type:
  type−specifier

atomic−statement:
  atomic *statement*

begin−statement:
  begin *statement*

binary−expression:
  expression binary−operator expression

binary−operator: *one of*
  + − ∗ / % ∗∗ & | ˆ << >> && || == != <= >= < > **by** #

block−statement:
  { *statements*$_{opt}$ }

bounded−range−literal:
  expression .. *expression*

break−statement:
  break *identifier*$_{opt}$;

call−expression:
  expression (*named−expression−list*)
  expression [*named−expression−list*]
  parenthesesless−function−identifier

cast−expression:
  expression : *type−specifier*

class−declaration−statement:
  class *identifier class−inherit−list*$_{opt}$ {
    class−statement−list$_{opt}$ }

class−inherit−list:
  : *class−type−list*

class−statement−list:
  class−statement
  class−statement class−statement−list

class−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

class−type−list:
  class−type
  class−type , *class−type−list*

class−type:
  identifier
  identifier (*named−expression−list*)

cobegin−statement:
  cobegin *block−statement*

coforall−statement:
  coforall *index−var−declaration* **in** *iterator−expression* **do** *statement*
  coforall *index−var−declaration* **in** *iterator−expression* *block−statement*
  coforall *iterator−expression* **do** *statement*
  coforall *iterator−expression* *block−statement*

conditional−statement:
  if *expression* **then** *statement* *else−part*$_{opt}$
  if *expression* *block−statement* *else−part*$_{opt}$

continue−statement:
  continue *identifier*$_{opt}$;

dataparallel−type:
  range−type
  domain−type
  mapped−domain−type
  array−type
  index−type

default−expression:
  = *expression*

do−while−statement:
  do *statement* **while** *expression* ;

domain−expression:
  domain−literal
  expression

domain−literal:
  [*range−expression−list*]

domain−map−expression:
  expression

domain−type:
  root−domain−type
  subdomain−type

else−part:
  else *statement*

empty−statement:
  ;

enum−constant−expression:
  enum−type . *identifier*

enum−constant−list:
  enum−constant
  enum−constant , *enum−constant−list*

enum−constant:
  identifier init−part$_{opt}$

enum−declaration−statement:
  enum *identifier* { *enum−constant−list* } ;

enum−type:
  identifier

expression−list:
  expression
  expression , *expression−list*

expression−statement:
  expression ;

expression:
  literal−expression
  variable−expression
  enum−constant−expression
  member−access−expression
  call−expression
  query−expression
  cast−expression
  lvalue−expression
  parenthesized−expression
  unary−expression
  binary−expression
  let−expression
  if−expression
  for−expression
  forall−expression
  reduce−expression
  scan−expression
  module−access−expression
  tuple−expression
  tuple−expand−expression
  locale−access−expression
  mapped−domain−expression

for−expression:
  for *index−var−declaration* **in** *iterator−expression* **do** *expression*
  for *iterator−expression* **do** *expression*

for−statement:
  for *index−var−declaration* **in** *iterator−expression* **do** *statement*
  for *index−var−declaration* **in** *iterator−expression block−statement*
  for *iterator−expression* **do** *statement*
  for *iterator−expression block−statement*

forall−expression:
  forall *index−var−declaration* **in** *iterator−expression* **do** *expression*
  forall *iterator−expression* **do** *expression*
  [*index−var−declaration* **in** *iterator−expression*] *expression*
  [*iterator−expression*] *expression*

forall−statement:
  forall *index−var−declaration* **in** *iterator−expression* **do** *statement*
  forall *index−var−declaration* **in** *iterator−expression block−statement*
  forall *iterator−expression* **do** *statement*
  forall *iterator−expression block−statement*
  [*index−var−declaration* **in** *iterator−expression*] *statement*
  [*iterator−expression*] *statement*

formal−intent: *one of*
  in out inout param type

formal−type:
  : *type−specifier*
  : ? *identifier*$_{opt}$

formal:
  formal−intent$_{opt}$ *identifier formal−type*$_{opt}$ *default−expression*$_{opt}$
  formal−intent$_{opt}$ *identifier formal−type*$_{opt}$ *variable−argument−expression*
  formal−intent$_{opt}$ *tuple−grouped−identifier−list formal−type*$_{opt}$ *default−expression*$_{opt}$
  formal−intent$_{opt}$ *tuple−grouped−identifier−list formal−type*$_{opt}$ *variable−argument−expression*

formals:
  formal
  formal , *formals*

function−body:
  block−statement
  return−statement

function−declaration−statement:
  def *function−name argument−list*$_{opt}$ *var−param−type−clause*$_{opt}$ *where−clause*$_{opt}$
    function−body

function−name:
  identifier
  operator−name

identifier−list:
  identifier−list , *tuple−grouped−identifier−list*
  identifier−list , *identifier*
  tuple−grouped−identifier−list
  identifier

if−expression:
  if *expression* **then** *expression* **else** *expression*
  if *expression* **then** *expression*

index−type:
  index (*domain−expression*)

index−var−declaration:
  identifier
  tuple−grouped−identifier−list

init−part:
  = *expression*

initialization−part:
  = *expression*

integer−parameter−expression:
  expression

irregular−domain−type:
  associative−domain−type
  opaque−domain−type

iterator−expression:
  expression

label−statement:
  label *identifier statement*

let−expression:
  let *variable−declaration−list* **in** *expression*

literal−expression:
  bool−literal
  integer−literal
  real−literal
  imaginary−literal
  string−literal
  range−literal
  domain−literal

locale−access−expression:
  expression . **locale**

locality−type:
  locale

lvalue−expression:
  variable−expression
  member−access−expression
  call−expression

mapped−domain−expression:
  domain−expression **dmapped** *domain−map−expression*

mapped−domain−type:
  domain−type **dmapped** *domain−map−expression*

member−access−expression:
  expression . *identifier*

method−declaration−statement:
  def *param−clause*$_{opt}$ *type−binding function−name argument−list*$_{opt}$ *var−param−type−clause*$_{opt}$
    return−type$_{opt}$ *where−clause*$_{opt}$ *function−body*

module−access−expression:
  module−identifier−list . *identifier*

module−declaration−statement:
  module *module−identifier block−statement*

module−identifier−list:
  module−identifier
  module−identifier . *module−identifier−list*

module−identifier:
  identifier

module−name−list:
  module−name
  module−name , *module−name−list*

module−name:
  identifier
  module−name . *module−name*

named−expression−list:
  named−expression
  named−expression , *named−expression−list*

named−expression:
  expression
  identifier = *expression*

on−statement:
  on *expression* **do** *statement*
  on *expression block−statement*

opaque−domain−type:
  domain (**opaque**)

operator−name: *one of*
  + − ∗ / % ∗∗ ! == <= >= < > << >> & | ˆ ˜

param−clause:
  param

param−for−statement:
  for param *identifier* **in** *param−iterator−expression* **do** *statement*
  for param *identifier* **in** *param−iterator−expression block−statement*

param−iterator−expression:
  range−literal
  range−literal **by** *integer−literal*

parenthesesless−function−identifier:
  identifier

parenthesized−expression:
  (*expression*)

primitive−type−parameter−part:
  (*integer−parameter−expression*)

primitive−type:
  bool *primitive−type−parameter−part*$_{opt}$
  int *primitive−type−parameter−part*$_{opt}$
  uint *primitive−type−parameter−part*$_{opt}$
  real *primitive−type−parameter−part*$_{opt}$
  imag *primitive−type−parameter−part*$_{opt}$
  complex *primitive−type−parameter−part*$_{opt}$
  string

query−expression:
  ? *identifier*$_{opt}$

range−expression−list:
  range−expression
  range−expression, *range−expression−list*

range−expression:
  expression

range−literal:
  bounded−range−literal
  unbounded−range−literal

range−type:
  range (*named−expression−list*)

record−declaration−statement:
  record *identifier record−inherit−list*$_{opt}$ {
    record−statement−list }

record−inherit−list:
  : *record−type−list*

record−statement−list:
  record−statement
  record−statement record−statement−list

record−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

record−type−list:
  record−type
  record−type , *record−type−list*

record−type:
  identifier
  identifier (*named−expression−list*)

reduce−expression:
  reduce−scan−operator **reduce** *expression*
  class−type **reduce** *expression*

reduce−scan−operator: *one of*
  + ∗ && || & | ˆ **min max minloc maxloc**

regular−domain−type:
  domain (*named−expression−list*)

reindexing−expression:
  [*domain−expression*]

remote−variable−declaration−statement:
  on *expression variable−declaration−statement*

return−statement:
  return *expression*$_{opt}$;

return−type:
  : *type−specifier*

root−domain−type:
  regular−domain−type
  irregular−domain−type

scan−expression:
  reduce−scan−operator **scan** *expression*
  class−type **scan** *expression*

select−statement:
  select *expression* { *when−statements* }

serial−statement:
  serial *expression* **do** *statement*
  serial *expression block−statement*

single−type:
  single *type−specifier*

statement:
  block−statement
  expression−statement
  assignment−statement
  swap−statement
  conditional−statement
  select−statement

while−do−statement
do−while−statement
for−statement
label−statement
break−statement
continue−statement
param−for−statement
use−statement
type−select−statement
empty−statement
return−statement
yield−statement
module−declaration−statement
function−declaration−statement
method−declaration−statement
type−declaration−statement
variable−declaration−statement
remote−variable−declaration−statement
on−statement
cobegin−statement
coforall−statement
begin−statement
sync−statement
serial−statement
atomic−statement
forall−statement

statements:
statement
statement statements

structured−type:
class−type
record−type
union−type
tuple−type

subdomain−type:
sparse$_{opt}$ **subdomain** (*domain−expression*)

swap−operator:
<=>

swap−statement:
lvalue−expression swap−operator lvalue−expression

sync−statement:
sync *statement*

sync−type:
sync *type−specifier*

synchronization−type:
sync−type
single−type

tuple−expand−expression:
(... *expression*)

tuple−expression:
(*expression* , *expression−list*)

tuple−grouped−identifier−list:
(*identifier−list*)

tuple−type:
(*type−specifier* , *type−list*)

type−alias−declaration−list:
type−alias−declaration
type−alias−declaration , *type−alias−declaration−list*

type−alias−declaration−statement:
config$_{opt}$ **type** *type−alias−declaration−list* ;

type−alias−declaration:
identifier = *type−specifier*
identifier

type−binding:
identifier .

type−declaration−statement:
enum−declaration−statement
class−declaration−statement
record−declaration−statement
union−declaration−statement
type−alias−declaration−statement

type−list:
type−specifier
type−specifier , *type−list*

type−part:
: *type−specifier*

type−select−statement:
type select *expression−list* { *type−when−statements* }

type−specifier:
primitive−type
enum−type
locality−type
structured−type
dataparallel−type
synchronization−type

type−when−statement:
when *type−list* **do** *statement*
when *type−list* *block−statement*
otherwise *statement*

type−when−statements:
  type−when−statement
  type−when−statement type−when−statements

unary−expression:
  unary−operator expression

unary−operator: *one of*
  + − ~ !

unbounded−range−literal:
  expression ..
  .. *expression*
  ..

union−declaration−statement:
  union *identifier* { *union−statement−list* }

union−statement−list:
  union−statement
  union−statement union−statement−list

union−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

union−type:
  identifier

use−statement:
  use *module−name−list* ;

var−param−type−clause:
  var *return−type*$_{opt}$
  const *return−type*$_{opt}$
  param *return−type*$_{opt}$
  type

variable−argument−expression:
  ... *expression*
  ... ? *identifier*$_{opt}$
  ...

variable−declaration−list:
  variable−declaration
  variable−declaration−list , *variable−declaration*

variable−declaration−statement:
  config$_{opt}$ *variable−kind variable−declaration−list* ;

variable−declaration:
  identifier−list type−part$_{opt}$ *initialization−part*
  identifier−list type−part
  array−alias−declaration

variable−expression:
  identifier

variable−kind: *one of*
  param const var

when−statement:
  when *expression−list* **do** *statement*
  when *expression−list block−statement*
  otherwise *statement*

when−statements:
  when−statement
  when−statement when−statements

where−clause:
  where *expression*

while−do−statement:
  while *expression* **do** *statement*
  while *expression block−statement*

yield−statement:
  yield *expression* ;

A.3 Depth-First Lexical Productions

bool−literal: *one of*
  true false

identifier:
  letter legal−identifier−chars$_{opt}$
  _ *legal−identifier−chars*

letter: *one of*
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

legal−identifier−chars:
  legal−identifier−char legal−identifier−chars$_{opt}$

legal−identifier−char:
  letter
  digit
  $

digit: *one of*
  0 1 2 3 4 5 6 7 8 9

imaginary−literal:
  real−literal **i**
  integer−literal **i**

real−literal:
  digits$_{opt}$. *digits exponent−part*$_{opt}$
  digits exponent−part

digits:
  digit
  digit digits

exponent−part:
  e *sign*$_{opt}$ *digits*
  E *sign*$_{opt}$ *digits*

sign: *one of*
  + −

integer−literal:
  digits
  0x *hexadecimal−digits*
  0X *hexadecimal−digits*
  0b *binary−digits*
  0B *binary−digits*

hexadecimal−digits:
  hexadecimal−digit
  hexadecimal−digit hexadecimal−digits

hexadecimal−digit: *one of*
  0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

binary−digits:
  binary−digit
  binary−digit binary−digits

binary−digit: *one of*
  0 1

string−literal:
  " *double−quote−delimited−characters*$_{opt}$ "
  ' *single−quote−delimited−characters*$_{opt}$ '

double−quote−delimited−characters:
  string−character double−quote−delimited−characters$_{opt}$
  ' *double−quote−delimited−characters*$_{opt}$

string−character:
  any character except the double quote, single quote, or new line
  simple−escape−character
  hexadecimal−escape−character

simple−escape−character: *one of*
  \' \" \? \a \b \f \n \r \t \v

hexadecimal−escape−character:
  \x *hexadecimal−digits*

single−quote−delimited−characters:
  string−character single−quote−delimited−characters$_{opt}$
  " *single−quote−delimited−characters$_{opt}$*

A.4 Depth-First Syntax Productions

module−declaration−statement:
  module *module−identifier block−statement*

module−identifier:
  identifier

block−statement:
  { *statements$_{opt}$* }

statements:
  statement
  statement statements

statement:
  block−statement
  expression−statement
  assignment−statement
  swap−statement
  conditional−statement
  select−statement
  while−do−statement
  do−while−statement
  for−statement
  label−statement
  break−statement
  continue−statement
  param−for−statement
  use−statement
  type−select−statement
  empty−statement
  return−statement
  yield−statement
  module−declaration−statement
  function−declaration−statement
  method−declaration−statement
  type−declaration−statement
  variable−declaration−statement
  remote−variable−declaration−statement
  on−statement
  cobegin−statement
  coforall−statement
  begin−statement
  sync−statement
  serial−statement
  atomic−statement
  forall−statement

expression−statement:
  expression ;

expression:
  literal−expression
  variable−expression
  enum−constant−expression
  member−access−expression
  call−expression
  query−expression
  cast−expression
  lvalue−expression
  parenthesized−expression
  unary−expression
  binary−expression
  let−expression
  if−expression
  for−expression
  forall−expression
  reduce−expression
  scan−expression
  module−access−expression
  tuple−expression
  tuple−expand−expression
  locale−access−expression
  mapped−domain−expression

literal−expression:
  bool−literal
  integer−literal
  real−literal
  imaginary−literal
  string−literal
  range−literal
  domain−literal

range−literal:
  bounded−range−literal
  unbounded−range−literal

bounded−range−literal:
  expression .. *expression*

unbounded−range−literal:
  expression ..
  .. *expression*
  ..

domain−literal:
  [*range−expression−list*]

range−expression−list:
  range−expression
  range−expression, *range−expression−list*

range−expression:
  expression

variable−expression:
  identifier

enum−constant−expression:
  enum−type . *identifier*

enum−type:
  identifier

member−access−expression:
  expression . *identifier*

call−expression:
  expression (*named−expression−list*)
  expression [*named−expression−list*]
  parenthesesless−function−identifier

named−expression−list:
  named−expression
  named−expression , *named−expression−list*

named−expression:
  expression
  identifier = *expression*

parenthesesless−function−identifier:
  identifier

query−expression:
  ? *identifier*$_{opt}$

cast−expression:
  expression : *type−specifier*

type−specifier:
  primitive−type
  enum−type
  locality−type
  structured−type
  dataparallel−type
  synchronization−type

primitive−type:
  bool *primitive−type−parameter−part*$_{opt}$
  int *primitive−type−parameter−part*$_{opt}$
  uint *primitive−type−parameter−part*$_{opt}$
  real *primitive−type−parameter−part*$_{opt}$
  imag *primitive−type−parameter−part*$_{opt}$
  complex *primitive−type−parameter−part*$_{opt}$
  string

primitive−type−parameter−part:
  (*integer−parameter−expression*)

integer−parameter−expression:
  expression

locality−type:
  locale

structured−type:
  class−type
  record−type
  union−type
  tuple−type

class−type:
  identifier
  identifier (*named−expression−list*)

record−type:
  identifier
  identifier (*named−expression−list*)

union−type:
  identifier

tuple−type:
  (*type−specifier* , *type−list*)

type−list:
  type−specifier
  type−specifier , *type−list*

dataparallel−type:
  range−type
  domain−type
  mapped−domain−type
  array−type
  index−type

range−type:
  range (*named−expression−list*)

domain−type:
  root−domain−type
  subdomain−type

root−domain−type:
  regular−domain−type
  irregular−domain−type

regular−domain−type:
  domain (*named−expression−list*)

irregular−domain−type:
  associative−domain−type
  opaque−domain−type

associative−domain−type:
  domain (*associative−index−type*)

associative−index−type:
  type−specifier

opaque−domain−type:
  domain (**opaque**)

subdomain−type:
  sparse$_{opt}$ **subdomain** (*domain−expression*)

domain−expression:
  domain−literal
  expression

mapped−domain−type:
  domain−type **dmapped** *domain−map−expression*

domain−map−expression:
  expression

array−type:
  [*domain−expression*] *type−specifier*

index−type:
  index (*domain−expression*)

synchronization−type:
  sync−type
  single−type

sync−type:
  sync *type−specifier*

single−type:
  single *type−specifier*

lvalue−expression:
  variable−expression
  member−access−expression
  call−expression

parenthesized−expression:
  (*expression*)

unary−expression:
  unary−operator expression

unary−operator: *one of*
  + − ~ !

binary−expression:
  expression binary−operator expression

binary−operator: *one of*
  + − ∗ / % ∗∗ & | ˆ << >> && || == != <= >= < > **by** #

let−expression:
  let *variable−declaration−list* **in** *expression*

if−expression:
  if *expression* **then** *expression* **else** *expression*
  if *expression* **then** *expression*

for−expression:
  for *index−var−declaration* **in** *iterator−expression* **do** *expression*
  for *iterator−expression* **do** *expression*

forall−expression:
  forall *index−var−declaration* **in** *iterator−expression* **do** *expression*
  forall *iterator−expression* **do** *expression*
  [*index−var−declaration* **in** *iterator−expression*] *expression*
  [*iterator−expression*] *expression*

index−var−declaration:
  identifier
  tuple−grouped−identifier−list

tuple−grouped−identifier−list:
  (*identifier−list*)

identifier−list:
  identifier−list , *tuple−grouped−identifier−list*
  identifier−list , *identifier*
  tuple−grouped−identifier−list
  identifier

iterator−expression:
  expression

reduce−expression:
  reduce−scan−operator **reduce** *expression*
  class−type **reduce** *expression*

reduce−scan−operator: *one of*
  + ∗ && || & | ˆ **min max minloc maxloc**

scan−expression:
  reduce−scan−operator **scan** *expression*
  class−type **scan** *expression*

module−access−expression:
  module−identifier−list . *identifier*

module−identifier−list:
  module−identifier
  module−identifier . *module−identifier−list*

tuple−expression:
(*expression* , *expression−list*)

expression−list:
expression
expression , *expression−list*

tuple−expand−expression:
(... *expression*)

locale−access−expression:
expression . **locale**

mapped−domain−expression:
domain−expression **dmapped** *domain−map−expression*

assignment−statement:
lvalue−expression assignment−operator expression

assignment−operator: *one of*
= += −= ∗= /= %= ∗∗= &= |= ˆ= &&= ||= <<= >>=

swap−statement:
lvalue−expression swap−operator lvalue−expression

swap−operator:
<=>

conditional−statement:
if *expression* **then** *statement else−part*$_{opt}$
if *expression block−statement else−part*$_{opt}$

else−part:
else *statement*

select−statement:
select *expression* { *when−statements* }

when−statements:
when−statement
when−statement when−statements

when−statement:
when *expression−list* **do** *statement*
when *expression−list block−statement*
otherwise *statement*

while−do−statement:
while *expression* **do** *statement*
while *expression block−statement*

do−while−statement:
do *statement* **while** *expression* ;

for−statement:
  for *index−var−declaration* **in** *iterator−expression* **do** *statement*
  for *index−var−declaration* **in** *iterator−expression block−statement*
  for *iterator−expression* **do** *statement*
  for *iterator−expression block−statement*

label−statement:
  label *identifier statement*

break−statement:
  break $identifier_{opt}$;

continue−statement:
  continue $identifier_{opt}$;

param−for−statement:
  for param *identifier* **in** *param−iterator−expression* **do** *statement*
  for param *identifier* **in** *param−iterator−expression block−statement*

param−iterator−expression:
  range−literal
  range−literal **by** *integer−literal*

use−statement:
  use *module−name−list* ;

module−name−list:
  module−name
  module−name , *module−name−list*

module−name:
  identifier
  module−name . *module−name*

type−select−statement:
  type select *expression−list* { *type−when−statements* }

type−when−statements:
  type−when−statement
  type−when−statement type−when−statements

type−when−statement:
  when *type−list* **do** *statement*
  when *type−list block−statement*
  otherwise *statement*

empty−statement:
  ;

return−statement:
  return $expression_{opt}$;

yield−statement:
  yield *expression* ;

module−declaration−statement:
  module *module−identifier block−statement*

function−declaration−statement:
  def *function−name argument−list*$_{opt}$ *var−param−type−clause*$_{opt}$ *where−clause*$_{opt}$
    function−body

function−name:
  identifier
  operator−name

operator−name: *one of*
  + − ∗ / % ∗∗ ! == <= >= < > << >> & | ˆ ˜

argument−list:
  (*formals*$_{opt}$)

formals:
  formal
  formal , *formals*

formal:
  formal−intent$_{opt}$ *identifier formal−type*$_{opt}$ *default−expression*$_{opt}$
  formal−intent$_{opt}$ *identifier formal−type*$_{opt}$ *variable−argument−expression*
  formal−intent$_{opt}$ *tuple−grouped−identifier−list formal−type*$_{opt}$ *default−expression*$_{opt}$
  formal−intent$_{opt}$ *tuple−grouped−identifier−list formal−type*$_{opt}$ *variable−argument−expression*

default−expression:
  = *expression*

formal−intent: *one of*
  in out inout param type

formal−type:
  : *type−specifier*
  : ? *identifier*$_{opt}$

variable−argument−expression:
  ... *expression*
  ... ? *identifier*$_{opt}$
  ...

var−param−type−clause:
  var *return−type*$_{opt}$
  const *return−type*$_{opt}$
  param *return−type*$_{opt}$
  type

return−type:
  : *type−specifier*

where−clause:
  where *expression*

function–body:
block–statement
return–statement

method–declaration–statement:
def *param–clause*$_{opt}$ *type–binding function–name argument–list*$_{opt}$ *var–param–type–clause*$_{opt}$
return–type$_{opt}$ *where–clause*$_{opt}$ *function–body*

param–clause:
param

type–binding:
identifier .

type–declaration–statement:
enum–declaration–statement
class–declaration–statement
record–declaration–statement
union–declaration–statement
type–alias–declaration–statement

enum–declaration–statement:
enum *identifier* { *enum–constant–list* } ;

enum–constant–list:
enum–constant
enum–constant , *enum–constant–list*

enum–constant:
identifier init–part$_{opt}$

init–part:
= *expression*

class–declaration–statement:
class *identifier class–inherit–list*$_{opt}$ {
class–statement–list$_{opt}$ }

class–inherit–list:
: *class–type–list*

class–type–list:
class–type
class–type , *class–type–list*

class–statement–list:
class–statement
class–statement class–statement–list

class–statement:
type–declaration–statement
function–declaration–statement
variable–declaration–statement
empty–statement

record−declaration−statement:
  record *identifier record−inherit−list*$_{opt}$ {
    record−statement−list }

record−inherit−list:
  : *record−type−list*

record−type−list:
  record−type
  record−type , *record−type−list*

record−statement−list:
  record−statement
  record−statement record−statement−list

record−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

union−declaration−statement:
  union *identifier* { *union−statement−list* }

union−statement−list:
  union−statement
  union−statement union−statement−list

union−statement:
  type−declaration−statement
  function−declaration−statement
  variable−declaration−statement
  empty−statement

type−alias−declaration−statement:
  config$_{opt}$ **type** *type−alias−declaration−list* ;

type−alias−declaration−list:
  type−alias−declaration
  type−alias−declaration , *type−alias−declaration−list*

type−alias−declaration:
  identifier = *type−specifier*
  identifier

variable−declaration−statement:
  config$_{opt}$ *variable−kind variable−declaration−list* ;

variable−kind: *one of*
  param const var

variable−declaration−list:
  variable−declaration
  variable−declaration−list , *variable−declaration*

variable−declaration:
  identifier−list type−part$_{opt}$ *initialization−part*
  identifier−list type−part
  array−alias−declaration

initialization−part:
  = *expression*

type−part:
  : *type−specifier*

array−alias−declaration:
  identifier reindexing−expression$_{opt}$ => *array−expression* ;

reindexing−expression:
  [*domain−expression*]

array−expression:
  expression

remote−variable−declaration−statement:
  on *expression variable−declaration−statement*

on−statement:
  on *expression* **do** *statement*
  on *expression block−statement*

cobegin−statement:
  cobegin *block−statement*

coforall−statement:
  coforall *index−var−declaration* **in** *iterator−expression* **do** *statement*
  coforall *index−var−declaration* **in** *iterator−expression block−statement*
  coforall *iterator−expression* **do** *statement*
  coforall *iterator−expression block−statement*

begin−statement:
  begin *statement*

sync−statement:
  sync *statement*

serial−statement:
  serial *expression* **do** *statement*
  serial *expression block−statement*

atomic−statement:
  atomic *statement*

forall−statement:
  forall *index−var−declaration* **in** *iterator−expression* **do** *statement*
  forall *index−var−declaration* **in** *iterator−expression block−statement*
  forall *iterator−expression* **do** *statement*
  forall *iterator−expression block−statement*
  [*index−var−declaration* **in** *iterator−expression*] *statement*
  [*iterator−expression*] *statement*

Index