Chapel Language Specification 0.703 (internal)

Cray Inc
411 First Ave S, Suite 600
Seattle, WA 98104

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 Language Systems initiative and the DARPA High Productivity Computing Systems initiative.

This document specifies the Chapel language. It is a work in progress and is not definitive. In particular, it is not a standard.

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 italicised.

> *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:

- Section 1, Scope, describes the scope of this specification.
- Section 2, Notation, introduces the notation that is used throughout this specification.
- Section 3, Organization, describes the contents of each of the sections within this specification.
- Section 4, Acknowledgments, offers a note of thanks to people and projects.
- Section 5, Language Overview, describes Chapel at a high level.
- Section 6, Lexical Strucutre, describes the lexical components of Chapel.
- Section 7, Types, describes the types in Chapel and defines the primitive and enumerated types.
- Section 8, Variables, describes variables and constants in Chapel.
- Section 9, Conversions, describes the legal implicit and explict conversions allowed between values of different types. Chapel does not allow for user-defined conversions.
- Section 10, Expressions, describes the serial expressions in Chapel.
- Section 11, Statements, describes the serial statements in Chapel.
- Section 12, Modules, describes modules, Chapel's abstraction to allow for name space management.
- Section 13, Functions, describes functions and function resolution in Chapel.
- Section 14, Classes, describes reference classes in Chapel.
- Section 15, Records, describes records or value classes in Chapel.
- Section 16, Unions, describes unions in Chapel.
- Section 17, Tuples, describes tuples in Chapel.
- Section 18, Ranges, describes ranges in Chapel.
- Section 19, Domains and Arrays, describes domains and arrays in Chapel. Chapel arrays are more general than arrays in many other languages. Domains are index sets, an abstraction that is typically not distinguished from arrays.
- Section 20, Iterators, describes iterator functions and a class iterator interface in Chapel.
- Section 21, Generics, describes Chapel's support for generic functions and types.
- Section 22, Parallelism and Synchronization, describes parallel expressions and statements in Chapel as well as synchronization constructs and atomic sections.
- Section 23, Locality and Distribution, describes constructs for managing locality and distributing data in Chapel.
- Section 24, Reductions and Scans, describes the built-in reductions and scans as well as structural interfaces to support user-defined reductions and scans.

- Section 25, Input and Output, describes support for input and output in Chapel, including file input and output..
- Section 26, Standard Modules, describes the standard modules that are provided with the Chapel language.

4 Acknowledgments

We would like to recognize the following people for their efforts and impact on the Chapel language and its implementation—David Callahan, Hans Zima, John Plevyak, Shannon Hoffswell, Roxana Diaconescu, Mark James, Mackale Joyner, and Robert Bocchino.

Chapel is a derivative of a number of parallel and distributed languages and takes ideas directly from them. These include 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

5.1 Motivating Principles

Chapel is a new programming language being developed by Cray Inc. as part of DARPA's High Productivity Computing Systems (HPCS) program to improve the productivity of programming parallel systems. There are four main motivating principles for the design of the Chapel language.

General Parallel Programming Chapel's first motivating principle is to support general parallel programming through the use of high-level language abstractions for expressing parallelism. Chapel supports a *global-view programming model* that raises the level of abstraction for the expression of both data and control flow as compared to parallel programming models currently used in production.

In Chapel, a *locale* is an abstraction for the processing unit of the target architecture. Locales are to be defined such that a locale's threads will all have 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.

Global-view data structures are arrays and other data aggregates whose sizes and indices are expressed globally in spite of the fact that their implementations may distribute them across the memories of multiple locales. This contrasts with most parallel languages used in practice, which tend to require users to partition distributed data aggregates into per-processor chunks, either manually or using language abstractions. As a simple example, to create 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 $0 \dots n-1$. 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 inter-processor 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.

Control of Locality 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. We consider this control over program locality to be essential for achieving 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.

Object-Oriented Programming (OOP) A third principle in Chapel is support for object-oriented programming. OOP 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 OOP methodologies.

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.

Chapel's first two principles are designed to provide support for general, performance-oriented parallel programming through high-level abstractions. The second two principles are supported to help narrow the gulf that exists between parallel programming languages and mainstream programming and scripting languages.

5.2 Summary of Chapel Language Features

This following section provides a summary of Chapel language features. This summary is not intended to give a complete description of each feature. Rather, it should give the reader a map to the rest of the specification document.

> *TODO:* Need to add references to other sections in the spec so that this does function as a map to the rest of the document.
>
> Also add references to the examples in this section?

5.2.1 Chapel Programming Basics

Hello World A Chapel version of the standard "hello world" computation is given here:

```
writeln("Hello, world!"); //writes Hello, world! to stdout
```

This program contains a single line of code that makes a call to the standard `writeln` subroutine, passing it a string literal argument, `"Hello, world!"`. This call causes the string to be printed to the console when the program is executed.

Chapel Programs 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 subroutine 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, it need not define an entry point. Thus, when the program is executed, the single `hello` module will be initialized by executing its top-level code, invoking the call to `writeln()`, and printing out the message.

Comments There are two ways of providing comments in Chapel source code. Inserting "`//`" indicates that a comment follows, through to the end of that line. Using "`/*`" indicates the start of a comment that may span multiple lines. In this case, the comment must be explicitly ended with "`*/`".

```
// Comments on a single line
/* Comments
   between lines */
```

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

```
> chpl -o hello hello.chpl
> ./hello
```

The following output will be printed to the console:

```
> Hello, world!
```

5.2.2 Variables

There are three kinds of Chapel variables: `var`, `const` (runtime constants), `param` (compile time constants).

Variable Declarations Chapel is a strongly typed language. Each variable declaration must include a type specification or an assignment to an initial value from which the type can be inferred. Variable declarations include the kind of variable, a type specification, and an assignment to an initial value. The type specification can be omitted if the type can be inferred from the initial value. If the variable is not initialized in its declaration, then it contains a default initial value according to its type.

Example. The following code gives examples of variable declarations in Chapel.

```
var x: real = 1.0; //x is real, init to 1.0
var y, z: real; //y, z are real, init to 0.0

var n = 10; //n inferred to be int, init to 10
const size = n; //size is runtime constant int,
                //set to 10
```

Configuration Variables The keyword `config` may precede any global variable declaration, indicating that the variable may be set at compile time or runtime. To set configuration variables at runtime, use "`--`" followed by the variable name and value on the execution command line. To set configuration variables at compile time, use "`-s`" followed by the variable name and value on the compiler command line.

Example. The following code gives an example of a configuration runtime constant, `n` and a configuration compile time constant, `debug`.

```
config const n = 100;
config param debug:bool;
```

To set `n` at runtime, the following command is used.

```
> ./a.out --n=1000
```

To set `debug` at compile time, the following compile commmand and flag is used.

```
> chpl testprogram.chpl -s debug=true
```

Synchronization Variables Single variables are assigned once. Using a single variable before it is assigned, suspends the computation until another task assigns a value to it. Sync variables are similar to single variables, but permit multiple assignments to that variable. Sync variables have a *full* or *empty* state associated with them that is modified when read and written to and control access to these variables between threads of computation.

```
var x: single int;
var y: sync real;
```

5.2.3 Types

Chapel supports primitive numeric, logical and string types. In addition, Chapel provides many non-primitive types such as classes, records, ranges, domains and arrays. The paragraphs in this section describe all of the Chapel types.

Primitive Types Chapel provides primitive data types for signed and unsigned integers, real, imaginary and complex floating point numbers, booleans and strings. The following table provides information about the names of these data types in Chapel, and their default sizes and initial values. If the size of the integer and floating point data types are not specified, then the default size is used. When declaring variables, if an initial value is not supplied then the default initial value for that variable's type is used.

Type	Default Size	Types with Specified Sizes	Default Initial Value
`int`	32 bits	`int(8)`, `int(16)`, `int(32)`, `int(64)`	`0`
`uint`	32 bits	`uint(8)`, `uint(16)`, `uint(32)`, `uint(64)`	`0`
`real`	64 bits	`real(32)`, `real(64)`, `real(128)`	`0.0`
`imag`	64 bits	`imag(32)`, `imag(64)`, `imag(128)`	`0.0i`
`complex`	128 bits	`complex(64)`, `complex(128)`, `complex(256)`	`0.0 + 0.0i`
`bool`	1 bit		`false`
`string`	unbounded		`""`

Non-Primitive Types The non-primitive types are listed in the table below, along with their default initial values for variables of that type.

Type	Default Initial Value
`enum`	first enum constant
`class`	`nil`
`record`	default constructed record
`range`	empty range: `(1..0)`
`domain`	empty range for each dimension
`array`	default initial values of element types
`tuple`	default initial values of component types
`locale`	ordered list of processor ids assigned at job launch
`file`	`(filename = , path = ., mode = r)`

The paragraphs below give a brief description and example for each type.

Enumerated Types An enumerated type defines an ordered set of named constants. By default, these constants can be coerced to an integral value, starting with one for the first constant. An integral value may be assigned to one or more of the constants to override the default behavior.

Example.

```
enum day {sun, mon, tue, wed, thu, fri, sat};
var d: day = wed;

writeln(d, " is day number ", d:int, " of the week");
```

This example would output:

```
wed is day number 4 of the week
```

Classes Classes are data structures defined with fields and methods, and they are assigned by reference. A class may inherit fields and methods from another class.

An instance of a class is created by calling its constructor in a variable declaration. Each call to the constructor instantiates a new object of the class and returns a reference to the object. Chapel provides a default class constructor for each class.

Example. The following code gives an example of the circle class. The variable, `x`, is an instance of the circle class with a radius of `1.0`.

```
class circle {
  var radius: real;
  def area {
    param pi = 3.14;
    return pi*(radius**2);
  }
}
var x = circle(radius=1.0);
```

Records Records are similar to classes. They contain fields and methods and can inherit fields and methods from other records. A record, however, directly contains the data associated with the fields in the record. A record is not a reference to storage location as classes are. Thus, records are assigned by value.

Example. In the example code below, `p` and `q` are both variables of the record `point`. They are distinct storage locations that are updated independently of each other.

```
record point {
  var x,y: real;
}
var p = point(0.0,1.0);
var q = p;
```

Tuple Types Tuples are lightweight records with anonymous fields. The fields can be all of the same type or of multiple types, and they are indexed in the order in which they were declared.

Example. The following variable declarations give an example of a mixed tuple, `pt` and a homogeneous tuple `x`.

```
var pt: (int,real);
var x: 3*int;
```

Among their many uses, tuples may be used in the assignment to a complex variable.

```
var x, y: real;
var z: complex;
  ⋮
z = (x,y):complex;
```

Ranges Ranges represent bounded and unbounded strided sequences of integral types. Ranges can be specified by the literal expression, ”`low..high by stride`” where `low`, `high` and `stride` are integral expressions. The bounds of the range, `low` and `high`, can be left unspecified to indicate an unbounded range. The stride is also optional. If `by stride` is omitted, the default stride of the range is one.

Ranges are most commonly used in the definition of arithmetic domains and in iterator expressions.

Example. The following example shows the definition of the range `span1D` which is the ordered set `1` to `n`. This range is then used in the definition of the arithmetic two-dimensional domain, `D`.

```
var span1D: range = 1..n;
var D = [span1D,span1D];
```

Domains and Arrays In Chapel, the indices and values for a data set are maintained in two distinct types of data structures. Domains specify indices and arrays store the values for sets of data. Arrays and domains may be arithmetic, sparse, indefinite, enumerated or opaque.

An array must be specified with a domain in its variable declaration, and it is linked with this domain during execution. When a domain is modified by adding or removing indices, all arrays declared with this domain reflect this change. Values that can be indexed by both the old and new domains are preserved in the array. Values that can only be accessed with the old domain are lost, and values that can be accessed by only the new domain result in new entries in the array. A domain can be declared with a distribution function which indicates how the data values in arrays declared with this domain are to be distributed across processors.

Arrays are passed by reference to functions where they may be modified, but assignment of arrays is by value.

Example. In the code below, `D2` is declared with the type `domain(2)` indicating that it is a two-dimensional domain. The domain declaration also includes an intialization for the domain using the range `1..n`, indicating that `D2` is an arithmetic domain. The arrays declared with `D2`, `A`, `B`, `C` are thus two-dimensional arithmetic arrays.

```
const D2: domain(2) = [1..n, 1..n];
var A, B, C: [D2] real;
```

The code below shows the declaration for a distributed two-dimensional arithmetic domain and arrays.

```
const DistD2: domain(2) distributed(Block) = [1..n, 1..n];
var A, B, C: [DistD2] real;
```

The Locale Type In Chapel, the term *locale* refers to the processing unit in a parallel computer system. The `locale` type is used to associate computations and data to these processing units. Chapel provides a predefined array `Locales` where each entry is of the `locale` type. User-defined variables of `locale` type can be declared to store entries from this `Locales` array or the entries can be accessed directly to indicate where data should be reside in memory or where computations should be executed.

The File Type Chapel provides a built-in `file` type to handle input and output to files. Variables declared to be of this type store information about the files that are to be accessed or created during execution, and they provide methods for reading and writing to these files. Variables of the `file` type contain the fields: `filename`, `path`, and `mode`. And, they provide the following methods: `open`, `close`, `read`, `write` and `writeln`.

The Type Alias A type alias may defined for any type. When used, it has the same meaning as using the aliased type directly.

```
type elemType = real(32);
var alpha: elemType; //alpha is 32-bit real
```

5.2.4 Operators

Operator Precedence

Operators	Associativity	Use
`.`	left	member access
`() []`	left	function call, index expression
`**`	right	exponentiation
unary `+ - ~`	right	sign and bitwise negation
`:`	left	cast
`* / %`	left	multiply, divide, and modulus
`+ -`	left	plus and minus
`&`	left	bitwise and
`^`	left	bitwise xor
`<< >>`	left	shift left and shift right
`|`	left	bitwise or
`<= >= < >`	left	ordered comparison
`== !=`	left	equality comparison
`!`	right	logical negation
`&&`	left	logical and
`||`	left	logical or
`..`	left	ranges
`in`	left	forall expressions
`by`	left	striding ranges and domains
`if`	left	conditional expressions
`reduce scan`	left	reductions and scans
`,`	left	comma separated expressions

Assignment Operators Supported assignment operators:

```
 = += -= *= /= %= **= &= |= ^= &&= ||= <<= >>=
```

The Swap Operator The swap operator, <=> swaps the values between the left and right-hand side expressions.

```
x <=> y; //swaps the value in x with the value in y
```

The Array Alias Operator The array alias operator, =>, creates an alias to an array or an array slice.

```
var A11 => A(block1,block1);
```

5.2.5 Serial Statements and Expressions

Block Statement A block statement is delimited by braces, { } and it contains a set of Chapel statements separated by semi-colons. Variables declared within a block statement are local to that block.

Example. The following code shows a block of statements for computing a Givens rotation.

```
{
tau = -a/b;
s = 1/sqrt(1 + tau*tau);
c = s*tau;
}
```

For Loop Statements In Chapel, a `for` loop statement defines an index and specifies either a function, expression or a variable to iterate over. For loops may be nested. They may also be defined with more than one index variable.

Example. The following code gives three examples of `for` loops.

```
for (i,j) in [1..n,1..n] {
  sum += x(i,j);
}

for index in userDefinedIterator(n) {
  sum += y(index);
}

for ij in D do sum += z(ij);
```

The first `for` loop defines a tuple of integers for the index of the loop and iterates over the tensor product promotion of the two arithmetic ranges from `1` to `n`. For more information about tensor product promotion see §20.4.2.

The second `for` loop demonstrates the use of an iterator function. The third `for` loop iterates over a domain D and shows the alternative definition of a `for` loop using the `do` keyword when the loop body is a single statement.

Conditional Statement The Chapel conditional statement uses the standard `if–then–else` structure. When the then clause is a block of statements, delimited with '{' and '}', the `then` keyword may be omitted.

Example.

```
if (a == 0.0) then
  x = 0.0;
else
  x = a*y;
if (b != 0.0) {
  z = b*x;
}
```

The example demonstrates two conditional statements, one with both `then` and `else` clauses, and one with just a `then` clause.

Select Statement Chapel provides a `select` statement that allows the option of executing different statement blocks, depending on the value of the select variable.

Example.

```
select pos {
  when "first" do x = y(1);
  when "last" do x = y(n);
  when "middle" do x = y(((n/2):int) + n%2);
  otherwise writeln("invalid position");
}
```

In the above example, the `select` statement specifies four possible statements to execute, depending on the value of the string, `pos`.

Type Select Statement The `type select` statement allows the choice of executing different statement blocks depending on the type of the select variable.

Example.

```
type select ind {
  when int do y = x(ind);
  when real, uint do y = x(ind:int);
  otherwise y = 0.0;
}
```

In the above example, the variable `ind` is used to index into the array `x` if it has the appropriate type.

TODO: New example in Quick Reference Guide will be replace this example as well.

Query Expression The query expression uses a `?` to indicate that the type of the preceding variable be queried and assigned to the following variable.

Example.

```
def sumOfThree(x: ?t, y:t, z:t):t {
   return x + y + z;
}
```

TODO: Move to function subsection.

Let Expression The `let–in` expression defines variables to be used only in the expression following the `in` keyword.

Example.

```
x = let t1 = sqrt(b*b - 4.0*a*c), t2 = 2.0*a
    in ((-b + t1)/t2, (-b - t1)/t2);
```

Conditional Expression Chapel provides a conditional expression that can be evaluated and then assigned.

Example.

```
var half = if (i % 2) then i/2 +1 else i/2;
```

5.2.6 Functions and Iterators

Function Definitions Functions are defined with the keyword `def`, a name and a list of function arguments. Optionally, intents and types for the function arguments, return types and a where clause can be specified.

The Chapel language supports function overloading. At compile time, function resolution will identify the most specific function for each function call from a set of candidate functions depending on the number of function arguments, the types specified for the input and return values, and any where clauses.

Example. The example below defines a simple function that computes the area of a rectangle. The `areaRect` function is defined to take two function arguments of type `real` and return a value of type `real`. If this function is called with function arguments that have a type other than `real`, it will result in a function resolution error at compile time.

```
def areaRect(x:real, y:real): real {
  return x*y;
}
```

Function Intents Intents of function arguments specify how an argument can be modified during and after the function call. The intents can be specified as `in`, `out` or `inout`.

If the intent is omitted, then the argument has a blank intent, copying in the value using the assignment operator. Arguments with a blank intent cannot be assigned within a function. If the intent of an argument is `in`, the actual argument is copied to the formal argument and it may be modified within the function. However, the actual argument at the call site does not reflect any modifications made within the function. The `out` intent indicates that 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 `inout` intent indicates that the actual argument be copied to the formal argument and then copied back after the call.

Actual arguments of array and domain types are handled differently when they have a blank intent. In this case, array and domain arguments are passed by reference. Thus, they can be modified within the function and those modifications are reflected at the call site. Arrays arguments with non-blank intents are handled according to definition, being copied to and from the formal arguments. Actual arguments of domain types can only have a blank intent.

TODO: This section may be replaced by a table. I will be creating a table that summarizes intents for the Quick Reference Guide and it may be substituted for the text here.

Generic Functions The Chapel language supports generic functions, allowing the user to define a function without specifying the types of the formal arguments or return variables. A generic function definition can have formal arguments of generic type, formal arguments without a specified type or with a queried type, or formal arguments that are tagged with `type` or `param` keywords. In addition, a function is considered to be generic if it contains an array argument with its domain or element type unspecified or queried.

Example.

```
def absSum(x:[?D]) {
  var sum: x.eltType;
  for i in D do sum += abs(x(i));
  return sum;
}
```

This routine, `absSum`, can be called with arrays of any element type for which an `abs` function is defined. Further, this routine can be called with arrays of any domain type, so it can be called with arrays of arbitrary dimension.

Iterators Iterators are defined similarly to functions, except that they are defined with the keyword `iterator` and they contain a `yield` statement. Iterators may be called in `for` and `forall` loop statements. For each iteration of the loop, a value from the iterator is yielded and the body of the loop executed.

Example.

```
iterator evens(n) {
  for i in 1..n do yield (i,2*i);
}

for (i,i2) in evens(m) {
  x(i) = y(i2);
}
```

The above code gives an example of an iterator definition and use. The iterator, `evens` yields a tuple of integers at each iteration.

5.2.7 Input and Output

File I/O The Chapel language provides a `file` type for use in reading and writing to files. To read or write from a file, a variable of `file` type must be declared and then assigned the name and path of the file and whether the file is to be read or written. Methods to open, to close, to read and to write from that file can be invoked with this file variable.

TODO: Add file reading/writing example.

Chapel Standard I/O Functions Chapel provides built-in functions `write` and `writeln` to write to `stdout`, and the built-in function `read` that reads from `stdin`.

Default I/O Methods for Chapel Types Default `read`, `write`, and `writeln` methods are provided for all Chapel types.

Example. The following code shows the definition of an arithmetic domain `D` and array `A`, and calling the `writeln` function to print both variables to standard output.

```
var D = [1..5, 1..5];
var A: [D] real;

writeln("Writing a variable of domain type:");
writeln(D);
writeln("Writing a variable of array type:");
writeln(A);
```

The output is shown below. Variables of arithmetic domain type are formatted to print the ranges for each dimension of the domain. Variables of arithmetic array type are formatted to print the values of each row separated by a carriage return.

```
Writing a variable of domain type:
[1..5, 1..5]
Writing a variable of array type:
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
```

5.2.8 Parallel Features

The Chapel language is designed ease the challenges of parallel programming. Since the language has specific constructs to support parallel computations, the user can easily express the parallel work, and the compiler can easily identify it.

Chapel supports both data parallelism and task parallelism. To efficiently manage both types of parallelism a multi-processor system, users can specify how data and work is to be distributed across processors by using distribution and locality features.

Status note. Currently, the Chapel compiler supports only single-locale features. Task parallel features will spawn computations only on one locale. Data parallel features execute sequentially on one locale.

More examples and discussion of parallel features will be added to the language specification when they are available in the compiler.

Data Parallelism Data parallelism refers to work where the same operations can be executed concurrently on a set of data. To support data parallel work, the Chapel language provides the parallel forall loop statement. The iterations of a `forall` loop execute concurrently, as determined by the compiler and the runtime library.

The Chapel language provides an alternative, short-hand notation for defining a parallel loop. This notation omits the `forall` keyword and uses brackets to delimit the rest of the loop statement which includes the index and iterator expressions. This notation is convenient to use when the loop body is a single statement.

The compiler will also parallelize whole array and domain statements, where possible.

Example. The following code demonstrates three different ways to assign, in parallel, the scaled elements of the array `B` to the elements of the array `A`.

```
D = [1..m, 1..n];
A, B: [D] real;
alpha: real;
⋮

//Three equivalent definitions of a parallel loop
forall ij in D do {
  A(ij) = alpha*B(ij);
}

[ij in D] A(ij) = alpha*B(ij);

A = alpha*B;
```

The first loop uses the `forall` loop statement. The second loop uses the short-hand bracketed notation and the third loop expresses the computation using whole-array operations. All three loops will execute concurrently in the same manner, as determined by the compiler and runtime library.

TODO: Add forall expression.

Task Parallelism The Chapel language also supports parallelism where different types of operations are being executed concurrently with different data sets. In this case, computations are spawned through the use of the `begin` or `co-begin` statements. Synchronization between these computations is managed through the use of `sync` and `single` variables.

The `begin` statement spawns a computation to execute a statement. Control continues simultaneously with the statement following the begin statement. A begin statement cannot contain break, yield, or return statements.

The `cobegin` statement is used to create parallelism within a block statement. All statements within the block statement are executed concurrently. Any variable declared within the cobegin statement is a `single` variable.

Example.

```
var done: sync bool;

begin
  while(!done) do work(x);

otherwork(x);
done = true;

cobegin {
  init(A);
  init(B);
}
```

TODO: Add readXX, etc. functions?

Suppressing Parallelism It is often necessary to suppress concurrency during sections of parallel programs. Chapel offers two constructs to accomplish this: the `ordered` keyword and the `serial` statement.

TODO: Add example.

Scans and Reductions Chapel supports operations that execute over multiple locales through the use of `scan` and `reduce` expressions. The language provides built-in `scan` and `reduce` operators.

TODO: Add example.

Data Distributions and Locality Chapel manages the distribution of data through the use of domains and distributions. A distribution is a mapping of a domain's indices to locales. When a domain is declared with a distribution, then any iteration over that domain or its associated arrays will execute in parallel across the locales according to how the data is distributed.

The `on` statement controls on which locale a computation or data should be placed.

TODO: Not sure what to include here for now.

5.3 Example Chapel Programs

TODO: Review discussion of example programs to see what can be deleted because of previous introduction of Chapel features.

To illustrate Chapel language features, four short example codes are presented and discussed. Each example highlights certain Chapel features, showing how they are used in the context of a program. These examples do not cover all of Chapel's features. They are intended to introduce the user to many of the basic serial features that are currently supported in the compiler and to demonstrate how to program with these features. As more features are supported in the Chapel compiler, more examples will be added to this section.

All examples in this section are included with the release of the Chapel compiler.

5.3.1 Jacobi Method

Description The following example Chapel program solves a system of finite difference equations for the Laplace equation using the Jacobi method. The program uses two-dimensional arrays, `X` and `XNew`, to store and calculate the approximate solution. At each iteration, the next approximation for the solution at each grid point, `XNew(i,j)` is calculated by computing the average of the four neighboring grid points, `X(i-1,j)`, `X(i,j-1)`, `X(i+1,j)`, and `X(i,j+1)`. After all entries in `XNew` are computed, `X` is assigned `XNew` and convergence is tested. If convergence has not been reached, the next approximation is calculated, and so on until the convergence test is met.

Chapel Features This program demonstrates how *arrays* are declared and used. In Chapel, arrays are declared using *domains* . Domains are sets of indices which may be distributed across multiple processors indicating how data and parallel work should be divided among the processors. An array is a mapping from the domain to a collection of variables. An array is thus defined using a domain, distributing the entries of the array according to the domain's distribution. Domains may also be used to define array *slices* and as *iterators*. There are five kinds of domains in the Chapel language: *arithmetic*, *sparse*, *indefinite*, *opaque* and *enumerated*. This example program uses arithmetic domains and arrays.

Code Listing The Chapel code for this example follows.

```
config var n = 5,                         // size of nxn grid
           epsilon = 0.00001,             // convergence tolerance
           verbose = false;               // control for amount of output

def main() {
  const ProblemSpace = [1..n, 1..n],   // domain for interior points
        BigDomain = [0..n+1, 0..n+1];  // domain with boundary points

  var X, XNew: [BigDomain] real = 0.0; // X holds approximate solution
                                       // XNew is work array

  X[n+1, 1..n] = 1.0;

  if (verbose) {
    writeln("Initial configuration:");
    writeln(X, "\n");
  }

  var iteration = 0,                  // iteration counter
      delta: real;                    // covergence measure

  do {
    forall (i,j) in ProblemSpace do
      XNew(i,j) = (X(i-1,j) + X(i+1,j) + X(i,j-1) + X(i,j+1)) / 4.0;

    delta = max reduce abs(XNew[ProblemSpace] - X[ProblemSpace]);
    X[ProblemSpace] = XNew[ProblemSpace];

    iteration += 1;

    if (verbose) {
      writeln("iteration: ", iteration);
      writeln(X);
      writeln("delta: ", delta, "\n");
    }
  } while (delta > epsilon);

  writeln("Jacobi computation complete.");
  writeln("Delta is ", delta, " (< epsilon = ", epsilon, ")");
  writeln("# of iterations: ", iteration);
}
```

Execution and Output Compiling and running this program gives the following output:

```
> ./a.out
Jacobi computation complete.
Delta is 9.92124e-06 (< epsilon = 1e-05)
# of iterations: 60
```

It is possible to run a different sized problem, to use a different convergence tolerance, or enable more output without recompiling this program. There are three variables defined in lines 1 - 3, `n`, `epsilon` and `verbose`, which are *configuration variables* and can be set at the time of program execution through command line switches. Executing the following command line sequence,

```
> ./a.out --verbose=true --n=2 --epsilon=0.01
```

results in overriding the default values for `verbose`, `n` and `epsilon`, producing the following output.

```
Initial configuration:
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 1.0 1.0 0.0

iteration: 1
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.25 0.25 0.0
0.0 1.0 1.0 0.0
delta: 0.25

iteration: 2
0.0 0.0 0.0 0.0
0.0 0.0625 0.0625 0.0
0.0 0.3125 0.3125 0.0
0.0 1.0 1.0 0.0
delta: 0.0625

iteration: 3
0.0 0.0 0.0 0.0
0.0 0.09375 0.09375 0.0
0.0 0.34375 0.34375 0.0
0.0 1.0 1.0 0.0
delta: 0.03125

iteration: 4
0.0 0.0 0.0 0.0
0.0 0.109375 0.109375 0.0
0.0 0.359375 0.359375 0.0
0.0 1.0 1.0 0.0
delta: 0.015625

iteration: 5
0.0 0.0 0.0 0.0
0.0 0.117188 0.117188 0.0
0.0 0.367188 0.367188 0.0
0.0 1.0 1.0 0.0
delta: 0.0078125

Jacobi computation complete.
Delta is 0.0078125 (< epsilon = 0.01)
# of iterations: 5
```

Implementation Details This example program begins with the declaration of the three configuration variables. Note that these variables do not contain a type in their declaration:

```
config var n = 5,
           epsilon = 0.00001,
           verbose = false;
```

Instead the types of these variables are inferred from their initial values: `n` is an integer, `epsilon` is a real type, and `verbose` is a boolean type. In Chapel, if a variable declaration contains an initialization expression, it is optional to include a type specification. More details about variable declarations are given in §8.1.

After the configuration variables are declared, the `main` function is defined. The first two lines of `main` define two arithmetic domains:

```
def main() {
  const ProblemSpace = [1..n, 1..n],
        BigDomain = [0..n+1, 0..n+1];
```

Both domains are declared to be `const`, indicating that the values for the domains remain constant during the execution of the program. They are defined using ranges, and are, thus, arithmetic domains. Because the domains are initialized in their declaration, it is not necessary to specify that they are of domain type. In this case, `: domain(2)` was omitted from both domain declarations. `BigDomain` is essentially the `ProblemSpace` domain with additional boundary rows and columns. By defining `ProblemSpace` to be the interior points of the larger grid, `BigDomain`, the Jacobi computation can be cleanly specified in one line, line 24, for just the interior points, eliminating the need to write special case computations for the boundary points.

After the domains are declared, the arrays are declared using `BigDomain`, and the last row of `X` is set to one.

```
  var X, XNew: [BigDomain] real = 0.0;

  X[n+1, 1..n] = 1.0;
```

Because they are defined using `BigDomain`, the arrays `X` and `XNew` are of size `n+2` × `n+2`. They are declared to be of type `real` and all elements of the arrays are initialized to zero. When an array is assigned a scalar value, as in the initialization to zero, each element of the array is assigned the scalar value. In the following line, the notation `n+1, 1..n` indicates a *slice* of the array `X` is to be assigned the value one. Each entry in this slice, which is the last row of `X`, is set to one.

Lines 14 - 17 print the initial configuration of the problem, if `verbose` is set to true.

```
  if (verbose) {
    writeln("Initial configuration:");
    writeln(X, "\n");
  }
```

The `writeln` function outputs strings and values of variables that are passed as arguments, followed by a line return. When an array is passed as an argument, it is output is row-major order with linefeeds after each row. Including the character string, `"\n"` inserts an additional line return.

Lines 19 - 20 contain the remaining variable declarations:

```
  var iteration = 0,
      delta: real;
```

The variable `delta` must be specified to be of type `real` since it does not have an initialization expression.

The computational loop in lines 22 - 36 performs the Jacobi method. This loop is executed until `delta` is less than or equal to `epsilon`. Each iteration of the `do while` loop computes the next approximate solution using a `forall` loop.

```
    do{
      forall (i,j) in ProblemSpace do
        XNew(i,j) = (X(i-1,j) + X(i+1,j) + X(i,j-1) + X(i,j+1)) / 4.0;
```

This loop uses the `ProblemSpace` domain as an *iterator*. Iterators for loops provide a value or a set of values to be used in each iteration of the loop. In this case, each iteration is indexed by a *tuple* of indices `(i,j)`, which avoids the use of a nest of two loops, one loop for the index `i` and one loop for the index `j`. The compiler determines how this loop is made parallel, according to the default distribution for the `ProblemSpace` domain. Note that the `ProblemSpace` domain is the set of interior points of `X`'s domain. All of the references to `X` in this loop are defined, and do not go out of bounds.

After the next approximate solution is computed and stored in `XNew`, the change between `XNew` and `X`, for just the interior points, is calculated and stored in `delta`:

```
      delta = max reduce abs(XNew[ProblemSpace] - X[ProblemSpace]);
```

The *built-in reduction*, `max reduce` computes the maximum value of the expression that follows it. In this case the expression is the computed array of absolute values of the difference between the `XNew` and `X` arrays. If `XNew` and `X` are distributed, then this reduction is computed in parallel accordingly. The use of `ProblemSpace` with these two arrays indicates that the slice of the arrays corresponding to the interior points are to be used in this calculation.

Next, `X` is updated with the new approximate solution and the iteration number is advanced.

```
      X[ProblemSpace] = XNew[ProblemSpace];

      iteration += 1;
```

The domain `ProblemSpace` is used in the assignment of `XNew` to `X`, updating only the slice of `X` corresponding to the interior points of the array.

The remaining part of the loop optionally provides output about the iteration.

```
      if (verbose) {
        writeln("iteration: ", iteration);
        writeln(X);
        writeln("delta: ", delta, "\n");
      }
    } while (delta > epsilon);
```

If `verbose` is set to true, then information about the current approximate solution and `delta` is output. Then, `delta` is compared to the convergence tolerance, `epsilon`. If `delta` is not small enough, then the `do while` loop continues. Otherwise, the loop exits.

The program ends with three lines of output.

```
    writeln("Jacobi computation complete.");
    writeln("Delta is ", delta, " (< epsilon = ", epsilon, ")");
    writeln("# of iterations: ", iteration);
  }
```

Information about the convergence is given, printing the values for `epsilon` and the iteration counter.

5.3.2 Matrix and Vector Norms

Description The following example contains a module `Norm` which provides `norm` functions that compute either a vector or matrix norm, depending on the rank of the array that is passed as the argument. There are four norm type options: 1-norm, 2-norm, infinity norm and Frobenius norm. For vectors, all four options are implemented in this module. For matrices, all options but the 2-norm are provided. The module uses a variable of enumerated type `normType` to indicate the choice of norm.

Chapel Features This example demonstrates the definition and use of modules, generic functions and function overloading . When the `Norm` module is used, the user may call `norm(x)` or `norm(x,normType)` where `x` is any array and `normType` is the *enumerated type* as defined in the `Norm` module. The `norm` function is *overloaded* with four separate function definitions for `norm`, based on the the rank of the input array, or the number of formal arguments used in the `norm` function call. Each of these four function definitions is a *generic* function, not specifying the type of the array argument `x`. Generic functions allow for code reuse and readability.

Program Listing The following program gives the definition of the `Norm` module followed by a module which tests `Norm`, demonstrating different calls to the `norm` function.

```
module Norm{
  enum normType {norm1, norm2, normInf, normFrob};

  def norm(x: [], p: normType) where x.rank == 1 {
  // vector norm routine
    select (p) {
      when norm1 do return + reduce abs(x);
      when norm2 do return sqrt(+ reduce (abs(x)*abs(x)));
      when normInf do return max reduce abs(x);
      when normFrob do return sqrt(+ reduce (abs(x)*abs(x)));
      otherwise halt("Unexpected norm type");
    }
  }

  def norm(x: [?D], p: normType) where x.rank == 2 {
  // matrix norm routine
    select (p) {
      when norm1 do
        return max reduce [j in D(2)] (+ reduce abs(x[D(1), j]));

      when norm2 do
        halt("Haven't implemented 2-norm for 2D arrays yet");

      when normInf do
        return max reduce [i in D(1)] (+ reduce abs(x[i, D(2)]));

      when normFrob do return sqrt(+ reduce (abs(x)*abs(x)));

      otherwise halt("Unexpected norm type");
    }
  }

  def norm(x: [], p: normType) where x.rank > 2 {
    compilerError("Norms not implemented for array ranks > 2D");
  }

  def norm(x: []) {
  // default norm routine
39      select (x.rank) {
40        when 1 do return norm(x, norm2);
41        when 2 do return norm(x, normFrob);
42        otherwise compilerError("Norms not implemented for array ranks > 2D");
43      }
44    }
45  }

47  module TestNorm {
48    use Norm;

50    def testNorm(arr: []) {
51      // test all possible norms of arr
52      var testType = if (arr.rank == 1) then "vector" else "matrix";
53      writeln("Test of ", testType, " norms.  Array = ");
54      writeln(arr);
55      writeln("1-norm = ", norm(arr, norm1));
56      if (arr.rank == 1) then
57        writeln("2-norm = " , norm(arr, norm2));
58      writeln("infinity norm = ", norm(arr, normInf));
59      writeln("frobenius norm = ", norm(arr, normFrob));
60      writeln("default norm = ", norm(arr));
61      writeln();
62    }

64    def main() {
65      // test vector norms:
66      const D1 = [1..4];
67      var a:[D1] real;
68      a = 2.0;
69      testNorm(a);

71      // test matrix norms:
72      const D2 = [1..2,1..2];
73      var b:[D2] real;
74      b = 2.0;
75      testNorm(b);
76    }
77  }
```

Execution and Output After the definition of the `Norm` module, a `TestNorm` module is defined in lines 47 - 77, giving an example of how the `norm` functions can be used in a program.

On line 48 , `use Norm`, indicates that the `Norm` module is to be used when resolving functions in the `TestNorm` module. A `testNorm` function is defined, taking `arr` as an argument. Based on `arr.rank`, all of the valid norm options are tested, along with the generic `norm(arr)` function call. This `testNorm` function is called to test two arrays, `a` and `b`. The array `a` is a one-dimensional array, as defined by the domain `D1`. The array `b` is a two-dimensional array, as defined by the domain `D2`. Each array has a total of four elements, and each is initialized to 2.0.

The output of this program is given below. The first set of norms is computed for `a` which is a vector and the second set is computed for `b` which is a matrix. Even though the vector and matrix in this example contain the same number of elements with the same values, some of the computed norms are different between the vector and the matrix. Thus, different implementations of the norm functions are used to compute the norms, depending on the whether the input is a vector or matrix.

```
Test of vector norms.  Array =
2.0 2.0 2.0 2.0
1-norm = 8.0
2-norm = 4.0
infinity norm = 2.0
frobenius norm = 4.0
default norm = 4.0

Test of matrix norms.  Array =
2.0 2.0
2.0 2.0
1-norm = 4.0
infinity norm = 4.0
frobenius norm = 4.0
default norm = 4.0
```

Implementation Details The `Norm` module begins with defining `normType` to be an enumerated type with constant values `norm1`, `norm2`, `normInf`, `normFrob`. For three of the four `norm` function definitions (see lines 4, 15, 33), one of the formal arguments is of `normType`, indicating which type of norm is to be computed. If `norm` is called without the `normType` argument, then the default norm for vectors is the 2-norm (see line 40) and the default norm for matrices is the Frobenius norm (see line 41).

The first `norm` function is defined for vectors. This definition begins with line 4:

```
def norm(x: [], p: normType) where x.rank == 1 {
```

The function definition contains a `where x.rank == 1` clause, indicating that this version of `norm` is to be used when the input array `x` has just one dimension, and is thus, a vector. The arguments for this version of `norm` are `x` of generic array type and `p` of `normType`. Since all norm types for the vector case can be computed with whole array operations, there is no need to specify a domain type for `x`. It is good practice to specify `x` as an array type by using `: []` since the `norm` function is only defined for arrays. The compiler can detect errors if other, non-array types are passed as arguments to `norm` .

The body of the vector version of the `norm` function is a `select` statement on `p` in lines 6 - 12.

```
select (p) {
      when norm1 do return + reduce abs(x);
      when norm2 do return sqrt(+ reduce (abs(x)*abs(x)));
      when normInf do return max reduce abs(x);
      when normFrob do return sqrt(+ reduce (abs(x)*abs(x)));
      otherwise halt("Unexpected norm type");
    }
```

For the first four cases of the select statement, when `p` is `norm1`, `norm2`, `normInf` or `normFrob` a reduction operator is used along with `abs(x)` to compute the value that is returned. The `abs` function is promoted over the `x` array, computing the absolute values of each entry of `x`. The 1-norm of a vector is the sum of the absolute values of the entries of `x`, computed with the sum reduction expression, `+ reduce abs(x)`. The 2-norm of a vector is the sum of the squares of the absolute values of the entries of `x`, computed with the sum reduction expression, `+ reduce (abs(x)*abs(x))`. The infinity norm of a vector is the maximum entry of `x` in absolute value, which can be computed with the maximum reduction expression `max reduce abs(x)`. The Frobenius norm of a vector is the same computation as the 2-norm of a vector.

The last case of the select statement, the otherwise clause, results in the program halting, outputting the string that is passed to the `halt` function indicating that `p` has an unexpected value.

The second `norm` function is defined for matrices. This definition begins with line 15:

```
def norm(x: [?D], p: normType) where x.rank == 2 {
```

This version of the function will be used when `x` is a matrix, that is the `rank` of the array `x` is `2`. The arguments for this version of `norm` are `x` of generic array type and `p` of `normType`. For this matrix version, the domain of `x` is needed to express the norm computations. So, the domain type is *queried* and set to `D` with the expression `[?D]`.

The body of the matrix version of `norm` is a select statement on `p` in lines 17 - 30. The `norm1` case, the 1-norm of a matrix, is the maximum absolute column sum of the matrix.

```
when norm1 do
  return max reduce [j in D(2)] (+ reduce abs(x[D(1), j]));
```

The return expression is the maximum reduction of a `forall` loop, which computes a sum reduction for each iteration. It uses `D(1)` and `D(2)`, which are the first and second dimension, respectively, of the domain `D`. The shorthand version of a `forall` loop is used, `[j in D(2)]`, indicating that for all `j in D(2)`, the sum of the absolute values across the rows, `+ reduce abs(x[D(1), j])`, is computed. Then, the maximum value of these `j` absolute column sums is computed with `max reduce`.

The `norm2` case, the 2-norm of a matrix, is not implemented. Since the 2-norm of a matrix A is the square root of the maximum eigenvalue of the matrix $A^T A$ (or $A^H A$ in the case where A is complex), this norm has not been included in this simple example code. This case will halt with a message indicating this unimplemented status.

The `normInf` case, the infinity norm of a matrix, is the maximum absolute row sum.

```
when normInf do
  return max reduce [i in D(1)] (+ reduce abs(x[i, D(2)]));
```

For all `i in D(1)`, the sum of the absolute values across columns, `+ reduce abs(x[i, D(2)])` is computed. Then the maximum value of these `i` absolute row sums is computed with `max reduce`.

The `normFrob` case which is the Frobenius norm of a matrix, is the same computation as the 2-norm of a vector.

```
when normFrob do return sqrt(+ reduce (abs(x)*abs(x)));
```

The Frobenius norm is the square root of the sum of the squares of absolute values of all entries in the matrix.

The final case of the select statement, the `otherwise` clause, halts with a message that the norm type is unexpected.

The third version of the `norm` function with arguments `x` and `p`, given in lines 33 - 35 is for the case where `x.rank > 2`. The module was designed to only compute norms of vectors and matrices. Calling the `norm` function with three-dimensional or higher arrays should give an error at compile time. In this case, an appropriate compiler error is given using the `compilerError` function.

```
compilerError("Norms not implemented for array ranks > 2D");
```

For more information about user-defined compiler errors, see §6.5.

The final version of the `norm` function is given in lines 37 - 44.

```
def norm(x: []) {
  select (x.rank) {
    when 1 do return norm(x, norm2);
    when 2 do return norm(x, normFrob);
    otherwise compilerError("Norms not implemented for array ranks > 2D");
  }
}
```

This version has one formal argument, `x`, allowing the user to omit the norm type. It calls the other `norm` functions with default values for `normType`. For vectors, the default norm is defined to be the 2-norm. For matrices, the default norm is defined to be the Frobenius norm. This version of `norm` is a select statement on `x.rank`. Depending on the rank of `x`, the `norm` function is called with the appropriate default norm type or a compiler error is given.

5.3.3 Simple Producer-Consumer Program

Description The following example demonstrates a simple producer and consumer program. The *producer computation* sets the value of a sync variable and the *consumer computation* prints the value of the same variable.

Chapel Features The program contains a `begin` statement to implement two concurrent computations, and a `sync` variable to coordinate between the two. Sync variables have extra state associated with them to indicate whether they are logically *full* or *empty*. A sync variable is intended to be accessed by multiple concurrent computations, each of which may change the contents and state of the variable. A sync variable can be read when its state is full. Attempts to read an empty sync variable from one computation will suspend execution until another computation changes its state to full. Once a sync variable is able to be read, its state is atomically set to empty. Conversely, a sync variable can be written only when its state is empty. Attempts to write to a full sync variable will suspend until its state is empty. Once the variable is written to, the state is atomically set to full. In addition, there are functions that read and write sync variables which override this default behavior. More information about sync variables and their functions is in §22.7.2 and §22.7.3.

Program Listing The Chapel code for this example is given below.

```
use Time;

config var numIterations: int = 5,
           sleepTime: uint = 2;

var s: sync int;

begin { // create consumer computation
  for c in 1..numIterations do
    writeln("consumer got ", s);
}

// producer computation
for p in 1..numIterations {
  sleep(sleepTime);
  s = p;
}
```

Execution and Output When this program executes, a consumer computation is created with the `begin` statement. However, the consumer computation cannot read `s` before it has been written to. So, execution begins with the producer computation which assigns the value `1` to `s` after calling the `sleep` function[1]. Once `s` has the value `1`, the consumer computation can read it and print it. Execution switches back and forth between the producer and consumer computation for `numIterations`.

Running the program with the default values for `numIterations` results in:

```
consumer got 1
consumer got 2
consumer got 3
consumer got 4
consumer got 5
```

Each line is printed after a delay of `sleepTime` seconds.

Since the variables `numIterations` and `sleepTime` are *configuration variables*, they can be reset at execution time. For example, to change the number of iterations to 10 and to change the number of seconds to sleep to 5, the following execution command may be used:

```
> a.out --numIterations=10 --sleepTime=5
```

Implementation Details The program uses the `sleep` function in line 15, which is provided in the standard Chapel module, `Time`. So, the first line, `use Time`, is needed to include the `Time` module when resolving function calls in this program.

Lines 3 - 4 give the declarations and initial default values for the two variables, `numIterations` and `sleepTime`. The `config` keyword in front of these two variable declarations indicates that these are configuration variables. Configuration variables can be set to override their default values at program execution time, through the use of command line switches.

In line 4, the variable `s` is declared to be a `sync` variable of type `int`. Since `s` is not initialized, its state is empty at the beginning of the program's execution. The variable `s` will be used to synchronize the exchange of data between the producer and consumer computations.

The remainder of the code defines the consumer and producer computations. The new computation that is created with the `begin` statement will be referred to as the consumer computation. The continuing computation will be referred to as the producer computation. Execution control switches between the producer and consumer computations as the state of the sync variable `s` changes when it is read and written to.

The `begin` statement in line 8 creates a new computation to execute the `for` loop in lines 9 - 10.

```
begin {
  for c in 1..numIterations do
    writeln("consumer got ", s);
}
```

[1]The call to the `sleep` function is used to mimic some amount of computational time. It is not necessary to use a sleep function when synchronizing between concurrent computations.

This loop is indexed by `c`, which iterates over the range `1..numIterations`. The variable `c` is a new variable defined for the scope of the loop. Its type is inferred to be integer from the integer range that follows. During each iteration of the `for` loop, the `writeln` function is called which outputs the string literal `"consumer got "` and the value of the sync variable `s`, followed by a line break. In order to print the value stored in `s`, the sync variable is read only when its state is full. Once `s` is successfully read, its state will be set to empty. The next iteration follows, and the consumer computation will suspend execution until the state of `s` is made full again. Since the consumer computation only reads `s`, it will be the producer computation that changes the states of `s` to full.

The producer computation executes the `for` loop in lines 14 - 17.

```
for p in 1..numIterations {
  sleep(sleepTime);
  s = p;
}
```

This loop is indexed by `p` which iterates over the same range as the consumer computation. Like the index `c`, `p` is inferred to be an integer. During each iteration of this `for` loop, the `sleep` function is called with the arguments`leepTime`. This `sleep` function is provided in the `Time` standard module, and it causes the producer computation to `sleepTime` seconds. After returning from the `sleep` function, the producer computation assigns `s` to be the iteration number `p`. Because `s` is a sync variable, this assigment is executed only when the state of `s` is empty. Once `s` is written, its state is changed to full and the next iteration of the producer loop follows. Since the producer computation only writes `s`, it will be the consumer computation that changes the state of `s` to empty allowing the next iteration to write its iteration number to `s`.

The program terminates after `numIterations` of both the consumer and producer loops.

5.3.4 Generic Stack Implementations

Description Two implementations of a generic stack type, `Stack`, are given below. The first defines `Stack` to be a linked list of `MyNode` objects. In the second implementation, `Stack` is an array. Both implementations of `Stack` define the methods, `push`, `pop` and `isEmpty`.

Chapel Features These examples demonstrate *classes* and *records*. Chapel classes and records are both structured data types containing fields and methods. Classes are reference types while records are value types. The examples also use the *unspecified type alias* `itemType` as the generic type for the items in the stack and define generic stack methods. To use either version, a type for `itemType` must be specified when `Stack` is instantiated. More information about type aliases can be found in §21.3.1. The array implementation of the generic stack demonstrates the association of arrays to domains . In this example, the array's size is doubled by reassigning its domain to one that is twice in size.

Sample Stack A simple example of how this generic stack can be used:

```
var stack1: Stack(string);
stack1.push("one");
stack1.push("two");
stack1.push("three");
writeln(stack1.pop());
writeln(stack1.pop());
writeln(stack1.pop());
```

In this simple example, the variable `stack1` is declared to be a stack of string type. By specifying `string` as an input, the default constructor for `Stack` will initialize the type alias for the generic stack to be a string. The strings, `"one"`, `"two"` and `"three"` are pushed on `stack1`, and then three items are popped off of `stack1`, in the reverse order from how they were put on the stack. The output of this example is:

```
three
two
one
```

Linked List Implementation This implementation of a generic stack defines a *class*, `MyNode` and a *record*, `Stack`. In the following code, the reference pointers of `MyNode` objects are used to point to the top of the stack and to point between items in the stack, thus implementing the stack as a linked list.

```
 1 class MyNode {
 2   type itemType;
 3   var item: itemType;
 4   var next: MyNode(itemType);
 5 }

 7 record Stack {
 8   type itemType;
 9   var top: MyNode(itemType);

11   def push(item: itemType) {
12     top = MyNode(itemType, item, top);
13   }

15   def pop() {
16     if isEmpty then
17       halt("attempt to pop an item off an empty stack");
18     var oldTop = top;
19     top = top.next;
20     return oldTop.item;
21   }

23   def isEmpty return top == nil;
24 }
```

Linked List Implementation Details The code begins with a definition of the `MyNode` class in lines 1 - 5.

```
class MyNode {
  type itemType;
  var item: itemType;
  var next: MyNode(itemType);
```

Objects of type `MyNode` are used to store the generic items in the stack as a linked list. There are three fields in `MyNode`: `itemType`, `item` and `next`. `MyNode` objects are instantiated within `Stack` to have the same `itemType` as the `Stack`. The `item` field holds the data and `next` is a pointer to the next `MyNode` object in the linked list.

The `Stack` record contains two fields:

```
record Stack {
  type itemType;
  var top: MyNode(itemType);
```

When `Stack` is instantiated, a type is specified for the type alias, `itemType`. The `top` field is a pointer to the top of the stack, which is a `MyNode` object of `itemType`. When the stack is first instantiated, `top` is set to nil. To add and remove items from the stack, the `push` and `pop` methods are used.

The `push` method is given in lines 11 - 13. This method adds an item to the top of the stack by resetting `top` to point to a new `MyNode` object. This new top object stores the added item and sets its `next` field to the previous `top` object of the stack.

```
def push(item: itemType) {
  top = MyNode(itemType, item, top);
}
```

The default constructor for `MyNode` is called to create a new object with `itemType` and `top` fields of the `Stack` instance on which `pop` is called, and the formal argument `item` of the `pop` function call. To reference fields of an instance of a structured type, `this` is used. In this case, `this.top` and `this.itemType` are implicit in the uses of `top` and `itemType`.

The `pop` method is given in lines 15 - 21.

```
def pop() {
  if isEmpty then
    halt("attempt to pop an item off an empty stack");
  var oldTop = top;
  top = top.next;
  return oldTop.item;
}
```

This method returns the item at the top of the stack and resets `top` to point to the next object in the stack. First, a call to the `Stack` method `isEmpty` is made to determine if the stack is empty. If it is, the program halts with a message indicating that an attempt was made to pop an item off of an empty stack. Otherwise, `top` is reset, and the appropriate item is returned. In this method, `this.isEmpty` and `this.top` are implicit when `isEmpty` and `top` are used. The memory of the `oldTop` object is freed, through automatic garbage collection.

The `isEmpty` method, which is used to check if the stack is empty in the `pop` method, is defined to be:

```
def isEmpty return top == nil;
```

Array Implementation The following implementation of a generic stack uses the array, `data`, to store the generic items in the stack, and the counter, `numItems`, to track the number of items in the stack and to indicate the index of the top item of the stack. The number of items in the stack and the size of `data` are checked when a new item is pushed on the stack. If necessary, the size of `data` is doubled. This example demonstrates how the size of an array is increased by increasing its domain.

```
record Stack {
  type itemType;
  var numItems: int = 0;
  var data: [1..2] itemType;

  def push(item: itemType) {
    var height = data.numElements;
    if numItems == height then
      data.domain = [1..height*2];
    data(numItems+1) = item;
    numItems += 1;
  }
14   def pop() {
15     if isEmpty then
16       halt("attempt to pop an item off an empty stack");
17     numItems -= 1;
18     return data(numItems+1);
19   }

21   def isEmpty return numItems == 0;
22 }
```

Array Implementation Details There are three fields defined for this `Stack` record, `itemType`, an integer variable, `numItems` and an array `data` of `itemType`. When first instantiated, the stack is empty and `numItems` is initialized to zero. The `data` array is declared with an anonymous one-dimensional domain, `[1..2]`.

The same three methods, `push`, `pop`, and `isEmpty` are defined for this array implementation. Like the linked list implementation, there are no explicit references to `this` when accessing fields and methods.

In `push`, lines 6 - 12, `item` is stored in the `data(numItems+1)` and the `numItems` counter is incremented.

```
def push(item: itemType) {
  var height = data.numElements;
  if numItems == height then
    data.domain = [1..height*2];
  data(numItems+1) = item;
  numItems += 1;
}
```

Before `item` can be added to the stack, the size of `data` must be checked to determine if the array's size needs to be increased to accomodate another item being added to the stack. The method `numElements` is predefined for arrays, returning the total number of elements in an array. The variable `height` is set to the total number of elements in `data`, which is the size of the current array allocated to store items in the stack. If the number of items in the stack, `numItems` equals `height`, then more storage in `data` is needed. To increase the size of an array, the size of its domain is increased. By reseting `data`'s domain to a domain of twice the size, the array `data` itself is now doubled in size, and more items can be pushed onto the stack.

In `pop`, the top item in the stack, as indicated by the `numItem` counter, is returned, if the stack is not empty.

```
def pop() {
  if isEmpty then
    halt("attempt to pop an item off an empty stack");
  numItems -= 1;
  return data(numItems+1);
}
```

The `pop` method first checks to see if the stack is empty. If it is, the program halts indicating that there was an attempt to pop an item off of an empty stack. Otherwise, the `numItems` counter is decremented and `data(numItems+1)` is returned as the popped item.

The `isEmpty` method checks to see if the stack is empty, that is if `numItems` equals zero.

```
def isEmpty return numItems == 0;
```

6 Lexical Structure

This section describes the lexical components of Chapel programs.

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, are ignored by the compiler. 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.

6.2 White Space

White-space characters are spaces, tabs, and new-lines. Aside from delimiting comments and tokens, they are ignored by the compiler.

6.3 Case Sensitivity

Chapel is a case sensitive language so identifiers that are identical except of the case of the characters are still different.

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 must start with a letter, lower-case or upper-case, or an underscore, and can include lower-case letters, upper-case letters, digits, and the underscore.

Example. The following are legal identifiers:

```
x, x1e, xt3, legalChapelIdentifier, legal_chapel_identifier
```

6.4.2 Keywords

The following keywords are reserved:

```
atomic      begin       bool        break       by
class       cobegin     complex     config      const
continue    def         distributed do          domain
else        enum        false       for         forall
goto        if          imag        in          int
inout       iterator    let         locale      module
nil         of          on          ordered     otherwise
out         param       pragma      real        record
reduce      return      scan        select      serial
single      sync        then        true        type
uint        union       use         var         when
where       while       yield
```

6.4.3 Literals

Literal values for primitive types are described in 7.1.7.

6.4.4 Operators and Punctuation

The following special characters are interpreted by the syntax of the language specially:

symbols	**use**
`= += -= *= /= **= %= &= |= ^= &&= ||= <<= >>=`	assignment
`..`	ranges
`...`	variable argument lists
`&& || !`	logical operators
`& | ^ ~ << >>`	bitwise operators
`== != <= >= < >`	relational operators
`+ - * / % **`	arithmetic operators
`:`	types
`;`	statement separator
`,`	expression separator
`.`	member access
`?`	query types
`" '`	string delimiters

6.4.5 Grouping Tokens

The following braces are part of the Chapel language:

braces	**use**
`( )`	parenthesization, function calls, and tuples
`[ ]`	domains, square tuples, forall expressions, and function calls
`{ }`	type scopes and blocks

6.5 User-Defined Compiler Errors

The special compiler error statement given by

> *compiler-error-statement*:
>   **compilerError** (*expression-list*) ;
>
> *expression-list*:
>   *expression*
>   *expression* , *expression-list*

invokes a compiler error if the function that the statement is located within may be called when the program is executed and the statement is not eliminated by parameter folding.

The compiler error is defined by the expression list which can contain string literals and types. The error points to the spot in the Chapel program where the function containing the *compiler-error-statement* is called from.

7 Types

Chapel is a statically typed language with a rich set of types. These include a set of predefined primitive types, enumerated types, classes, records, unions, tuples, ranges, domains, and arrays. This section defines the primitive types, enumerated types, and type aliases.

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
```

Classes are discussed in §14. Records are discussed in §15. Unions are discussed in §16. Tuples are discussed in §17. Ranges are discussed in §18. Domains and arrays are discussed in §19.

7.1 Primitive Types

The primitive types include the following types: `bool`, `int`, `uint`, `real`, `complex`, `imag`, `string`, and `locale`. These primitive types are defined in this section except for the `locale` type which is defined in §23.1.1.

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`.

The relational operators return values of `bool` type and the logical operators operate on values of `bool` type.

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). For example, an integer can be used as the condition in a conditional statement. It is converted to `false` if it is zero, and otherwise, it is converted to `true`.

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.

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)` and `real(64)`, and sometimes include `real(128)`.

Arithmetic over real values follows the IEEE 754 standard.

7.1.4 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)`, and sometimes include `complex(256)`.

The real and imaginary components can be accessed via the methods `re` and `im`. The type of these components is real.

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

7.1.5 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)`, and sometimes include `imag(128)`.

> *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.

7.1.6 The String Type

Strings are a primitive type designated by the symbol `string`. Their length is unbounded.

Characters in a string can be accessed via the `substring` method on strings. This method takes an integer i and returns the ith character in the string.

> *Example*. The first character of a string `s` can be selected by the method call `s.substring(1)`.

7.1.7 Primitive Type 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
  0 '*x*' *hexadecimal-digits*
  0 '*b*' *binary-digits*

digits:
  digit
  digit digits

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

hexadecimal-digits:
  hexadecimal-digit
  hexadecimal-digit hexadecimal-digits

hexadecimal-digits: *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

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 are necessary 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*$_{opt}$

exponent-part:
  e *sign*$_{opt}$ *digits*

sign: *one of*
  + -

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

Note that real literals require that a digit follow the decimal point. This is necessary to avoid an ambiguity in interpreting `2.e+2` that arises if a method called `e` is defined on integers.

Imaginary literals are designated by the following syntax:

imaginary-literal:
  real-literal **i**
  integer-literal **i**

A complex number is specified by adding or subtracting an imaginary literal with a real literal. Alternatively, a 2-tuple literal of expressions of integer or real type can be cast to a complex. These expressions can be literals, but do not need to be. To create a complex literal or parameter, they must be literals or parameters.

Example. The following codes represent the same complex literal:

`2.0i`, `0.0+2.0i`, `(0.0,2.0):complex`.

String literals are designated by the following syntax:

string-literal:
  ” *characters*$_{opt}$ ”
  ’ *characters*$_{opt}$ ’

characters:
  character
  character characters

character:
  any-character

Status note. Strings are currently restricted to ASCII characters. In a future version of Chapel, strings will be defined over alphabets to allow for more exotic characters.

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
```

An enumerated type defines a set of named constants. These are associated with parameters of integral type. Each enumerated type is a distinct type.

7.3 Class Types

The class type defines a type that contains variables and constants, called fields, and functions, called methods. Classes are defined in §14. The class type can also contain type aliases and parameters. Such a class is generic and is defined in §21.

7.4 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. The difference between classes and records is elaborated on in §15.

7.5 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 §16.

7.6 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 §17.

7.7 Range Types

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

7.8 Domain and Array Types

Domains are index sets. Arrays are types that contain a set of zero or more elements all of the same type. The elements are referenced via indices that are in the domain that the array is declared over. Domains and arrays are defined in §19.

7.9 Type Aliases

Type aliases are declared with the following syntax:

type-alias-declaration-statement:
  type *type-alias-declaration* ;

type-alias-declaration:
  identifier type-part$_{opt}$
  identifier type-part$_{opt}$, *type-alias-declaration*

type-part:
  = *type*

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

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 (§21.3.1).

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 , *variable-declaration-list*

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

identifier-list:
  identifier
  identifier , *identifier-list*

type-part:
  : *type*
  : *synchronization-type type*

initialization-part:
  = *expression*

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.

Multiple variables can be defined in the same *variable-declaration-list*. All variables defined in the same *identifier-list* are defined to have the same type and initialization expression.

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.

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	an empty range
arrays	elements are default values
tuples	components are default values

8.1.2 Local Type Inference

If the type is omitted from a variable declaration, the type of the variable becomes the type of the initialization expression.

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 is managed automatically as discussed in §14.9.

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 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.
- The conditional expression where the condition is a parameter and the then- and else-expressions are parameters.
- Call expressions of parameter functions. See §13.7.

There is an expectation that parameters will be expanded to more types and more operations, and that functions that return parameters will be introduced, in the future.

8.4.2 Runtime Constants

Constants, as opposed to parameters, 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.

Variables of class type that are constants are constant references. The fields of the class can be modified, but the variable always points to the object that it was initialized to reference.

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 during compilation time via compilation flags or other implementation dependent means.

Example. A configuration parameter is set via a compiler flag. It may be used to control the target that is being compiled. For example, the code

```
config param target: string = "XT3";
```

sets a string parameter `target` to `"XT3"`. This can be checked to compile different code for this target.

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 (§9.1).

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 integeral 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 `int(8)`, `int(16)`, `int(32)`, `int(64)`, `uint(8)`, `uint(16)`, `uint(32)`, `uint(64)`, or `string`
- From `int(8)` to `int(16)`, `int(32)`, `int(64)`, `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `int(16)` to `int(32)`, `int(64)`, `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `int(32)` to `int(64)`, `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `int(64)` to `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `uint(8)` to `int(16)`, `int(32)`, `int(64)`, `uint(16)`, `uint(32)`, `uint(64)`, `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `uint(16)` to `int(32)`, `int(64)`, `uint(32)`, `uint(64)`, `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `uint(32)` to `int(64)`, `uint(64)`, `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `uint(64)` to `real(32)`, `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `real(32)` to `real(64)`, `real(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`

- From `real(64)` to `real(128)`, `complex(128)`, `complex(256)`, or `string`
- From `real(128)` to `complex(256)` or `string`
- From `imag(32)` to `imag(64)`, `imag(128)`, `complex(64)`, `complex(128)`, `complex(256)`, or `string`
- From `imag(64)` to `imag(128)`, `complex(128)`, `complex(256)`, or `string`
- From `imag(128)` to `complex(256)` or `string`
- From `complex(64)` to `complex(128)`, `complex(256)`, or `string`
- From `complex(128)` to `complex(256)`, or `string`
- From `complex(256)` to `string`

The implicit numeric conversions do not result in any loss of information except for the conversions from any of the `int` and `uint` types to any of the `real` and `complex` types and from any of the `real`, `imag`, and `complex` types to `string` where there is a loss of precision.

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. It can also be implicitly converted to `string` where the string is the name of the enumerated constant.

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

9.2 Explicit Conversions

Explicit conversions require a cast in the code. Casts are defined in §10.5. 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. Forall expressions are described in §22.2.

The syntax for an expression is given by:

> *expression*:
> *literal-expression*
> *variable-expression*
> *member-access-expression*
> *call-expression*
> *query-expression*
> *cast-expression*
> *lvalue-expression*
> *unary-expression*
> *binary-expression*
> *let-expression*
> *if-expression*
> *forall-expression*

10.1 Literal Expressions

A literal value for any of the built-in types is a literal expression. These are defined where the type is defined. The list of literal values is 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 is itself an expression. The syntax of a variable expression is given by:

> *variable-expression*:
> *identifier*

10.3 Call Expressions

The syntax to call a function is given by:

call-expression:
  expression (*named-expression-list*)

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

named-expression:
  expression
  identifier = *expression*

A *call-expression* is resolved to a particular function according to the algorithm for function resolution described in §13.9.

A *named-expression* is an expression that may be optionally named. The optional *identifier* represents a named actual argument described in §13.4.1.

10.3.1 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.

10.3.2 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.

10.4 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*

Querying is restricted to querying the type of a formal argument, the element type of an 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.

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;
}
```

The type alias `t` is used to specify the element type of array `y`. Arrays passed to this function must have element type `y`. The body of the function iterates over the domain of `y` captured in variable `D` and assigns the value of argument `x` to each element in array `y`.

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

10.5 Casts

A cast is specified with the following syntax:

cast-expression:
  expression : *type*

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.

10.6 LValue Expressions

An *lvalue* is an expression that can be used on the left-hand side of an assignment statement, passed to a formal argument of a function that has `out` or `inout` intent, or 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 more relaxed than the definition above. For example, not all *call-expression*s are lvalues.

10.7 Operator Precedence and Associativity

The following table summarizes the precedence of operators and their associativity. Operators listed earlier have higher precedence than those listed later.

operators	**associativity**	**use**
`.`	left	member access
`() []`	left	function call, index expression
`**`	right	exponentiation
unary `+ - ~`	right	sign and bitwise negation
`:`	left	cast
`* / %`	left	multiply, divide, and modulus
`+ -`	left	plus and minus
`&`	left	bitwise and
`^`	left	bitwise xor
`<< >>`	left	shift left and shift right
`|`	left	bitwise or
`<= >= < >`	left	ordered comparison
`== !=`	left	equality comparison
`!`	right	logical negation
`&&`	left	logical and
`||`	left	logical or
`..`	left	ranges
`in`	left	forall expressions
`by`	left	striding ranges and domains
`if`	left	conditional expressions
`reduce scan`	left	reductions and scans
`,`	left	comma separated expressions

10.8 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*
  + - * / % ** & | ^ << >> && || == != <= >= < >

The operators are defined in subsequent sections.

10.9 Arithmetic Operators

This section describes the predefined arithmetic operators. These operators can be redefined over different types using operator overloading (§13.8).

All 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.

10.9.1 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: real(128)): real(128)
def +(a: imag(32)): imag(32)
def +(a: imag(64)): imag(64)
def +(a: imag(128)): imag(128)
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.

10.9.2 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: real(128)): real(128)
def -(a: imag(32)): imag(32)
def -(a: imag(64)): imag(64)
def -(a: imag(128)): imag(128)
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.

10.9.3 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: real(128), b: real(128)): real(128)

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

def +(a: complex(64), b: complex(64)): complex(64)
def +(a: complex(128), b: complex(128)): complex(128)
def +(a: complex(256), b: complex(256)): complex(256)

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(128), b: imag(128)): complex(256)
def +(a: imag(128), b: real(128)): complex(256)

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: real(128), b: complex(256)): complex(256)
def +(a: complex(256), b: real(128)): complex(256)

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)
def +(a: imag(128), b: complex(256)): complex(256)
def +(a: complex(256), b: imag(128)): complex(256)
```

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.

10.9.4 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: real(128), b: real(128)): real(128)

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

def -(a: complex(64), b: complex(64)): complex(64)
def -(a: complex(128), b: complex(128)): complex(128)
def -(a: complex(256), b: complex(256)): complex(256)

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(128), b: imag(128)): complex(256)
def -(a: imag(128), b: real(128)): complex(256)

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: real(128), b: complex(256)): complex(256)
def -(a: complex(256), b: real(128)): complex(256)

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)
def -(a: imag(128), b: complex(256)): complex(256)
def -(a: complex(256), b: imag(128)): complex(256)
```

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.

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

def *(a: complex(64), b: complex(64)): complex(64)
def *(a: complex(128), b: complex(128)): complex(128)
def *(a: complex(256), b: complex(256)): complex(256)

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(128), b: imag(128)): imag(128)
def *(a: imag(128), b: real(128)): imag(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: real(128), b: complex(256)): complex(256)
def *(a: complex(256), b: real(128)): complex(256)

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)
def *(a: imag(128), b: complex(256)): complex(256)
def *(a: complex(256), b: imag(128)): complex(256)
```

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.

10.9.6 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: real(128), b: real(128)): real(128)

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

def /(a: complex(64), b: complex(64)): complex(64)
def /(a: complex(128), b: complex(128)): complex(128)
def /(a: complex(256), b: complex(256)): complex(256)

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(128), b: imag(128)): imag(128)
def /(a: imag(128), b: real(128)): imag(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: real(128), b: complex(256)): complex(256)
def /(a: complex(256), b: real(128)): complex(256)

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)
def /(a: imag(128), b: complex(256)): complex(256)
def /(a: complex(256), b: imag(128)): complex(256)
```

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.

10.9.7 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.

10.9.8 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)
def **(a: real(128), b: real(128)): real(128)
```

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.

10.10 Bitwise Operators

This section describes the predefined bitwise operators. These operators can be redefined over different types using operator overloading (§13.8.

10.10.1 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.

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

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

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

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

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

It is a compile-time error to apply the bitwise or operator to a value of type `uint(64)` and a value of type `int(64)`, and vice versa.

10.10.4 Bitwise Xor Operators

The bitwise xor operators are predefined as follows:

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

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

It is a compile-time error to apply the bitwise xor operator to a value of type `uint(64)` and a value of type `int(64)`, and vice versa.

10.11 Shift Operators

This section describes the predefined shift operators. These operators can be redefined over different types using operator overloading (§13.8.

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.

10.12 Logical Operators

This section describes the predefined logical operators. These operators can be redefined over different types using operator overloading (§13.8).

10.12.1 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.

10.12.2 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 a.isTrue then b.isTrue else false
```

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

```
def bool.isTrue return this;
```

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

10.12.3 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 a.isTrue then true else b.isTrue
```

The method `isTrue` is predefined over bool type as described in §10.12.2. Overloading the logical or operator over other types is accomplished by overloading the `isTrue` method over other types.

10.13 Relational Operators

This section describes the predefined relational operators. These operators can be redefined over different types using operator overloading (§13.8.

10.13.1 Ordered Comparison Operators

The "less than" comparison operators are predefined 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: real(128), b: real(128)): bool
def <(a: imag(32), b: imag(32)): bool
def <(a: imag(64), b: imag(64)): bool
def <(a: imag(128), b: imag(128)): 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 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: real(128), b: real(128)): bool
def >(a: imag(32), b: imag(32)): bool
def >(a: imag(64), b: imag(64)): bool
def >(a: imag(128), b: imag(128)): 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 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: real(128), b: real(128)): bool
def <=(a: imag(32), b: imag(32)): bool
def <=(a: imag(64), b: imag(64)): bool
def <=(a: imag(128), b: imag(128)): 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 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: real(128), b: real(128)): bool
def >=(a: imag(32), b: imag(32)): bool
def >=(a: imag(64), b: imag(64)): bool
def >=(a: imag(128), b: imag(128)): bool
```

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

10.13.2 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: real(128), b: real(128)): bool
def ==(a: imag(32), b: imag(32)): bool
def ==(a: imag(64), b: imag(64)): bool
def ==(a: imag(128), b: imag(128)): bool
def ==(a: complex(64), b: complex(64)): bool
def ==(a: complex(128), b: complex(128)): bool
def ==(a: complex(256), b: complex(256)): 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 §15.3.

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)`.

10.14 Miscellaneous Operators

This section describes several miscellaneous operators. These operators can be redefined over different types using operator overloading (§13.8.

10.14.1 The String Concatenation Operator

The string concatenation operator is predefined as follows:

```
def +(a: string, b: string): string
```

The result is the concatenation of `a` followed by `b`.

Example. Since integers can be implicitly converted to strings, an integer can be appended to a string using the string concatenation operator. The code

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

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

10.14.2 The Arithmetic Domain By Operator

The operator `by` is predefined on arithmetic domains. It is described in §19.3.3.

10.14.3 The Range By Operator

The operator `by` is predefined on ranges. It is described in §18.1.

10.15 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.

10.16 Conditional Expressions

A conditional expression is given by the following syntax:

conditional-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 §22.2.1.

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
  conditional-statement
  select-statement
  while-do-statement
  do-while-statement
  for-statement
  param-for-statement
  return-statement
  yield-statement
  module-declaration-statement
  function-declaration-statement
  type-declaration-statement
  variable-declaration-statement
  use-statement
  type-select-statement
  empty-statement
  cobegin-statement
  begin-statement
  serial-statement
  atomic-statement
  on-statement

The declaration statements are discussed in the sections that define what they declare. Module declaration statements are defined in §12. Function declaration statements are defined in §13. Type declaration statements are defined in §7. Variable declaration statements are defined in §8. Return statements are defined in §13.2. Yield statements are defined in §20.2.

The *cobegin-statement* is defined in §22.3. The *begin-statement* is defined in §22.4. The *serial-statement* is defined in §22.6. The *atomic-statement* is defined in §22.9. The *on-statement* is defined in §23.2.1.

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 (§22.3).

11.2 Block Level Statements

A block level statement is a category of statement that is sometimes called for by the language syntax. A block level statement is given by

```
block-level-statement:
  block-statement
  conditional-statement
  select-statement
  while-do-statement
  for-statement
  param-for-statement
  return-statement
  yield-statement
  type-select-statement
  empty-statement
  cobegin-statement
  begin-statement
  serial-statement
  atomic-statement
  on-statement
```

Block level statements are part of the language to avoid the excessive and unnecessary use of curly brackets. For example, function bodies are not required to be blocks, but must be block level statements.

11.3 Expression Statements

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

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

11.4 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 right-hand side of the assignment operator is evaluated first; it can be any expression. The expression on the left hand side must be a valid lvalue (§10.6). It is evaluated second and then assigned the value.

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 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)`.

> *Status note*. Currently, there is no verification that the right-hand side expression can be assigned to the left-hand side expression.

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 (§14.2), records (§15.2.3), unions (§16.2), tuples (§17.3), ranges (§18.2), domains (§19.1.3), and arrays (§19.2.4) is discussed in these later sections.

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-level-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-level-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-level-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 the structural iterator interface. The syntax of the for loop is given by:

for-statement:
  for *index-expression* **in** *iterator-expression* **do** *statement*
  for *index-expression* **in** *iterator-expression block-level-statement*

index-expression:
  expression

iterator-expression:
  expression

The index-expression can be an identifier or a tuple of identifiers. The identifiers are declared to be new variables for the scope of this statement.

If the iterator-expression is a tuple, 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 square 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-for-statement:
  for param *identifier* **in** *range-literal* **do** *statement*
  for param *identifier* **in** *range-literal block-level-statement*

Parameter for loops are restricted to iteration over range literals the bounds of which must also be parameters. The loop is then unrolled for each iteration.

11.9 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*

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

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

It is an error for a variable, type or module to be defined both by a `use` statement and by a declaration in the same scope. Functions may be overloaded in this way.

11.10 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 §16.3. 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-level-statement*
  otherwise *statement*

expression-list:
  expression
  expression , expression-list

type-list:
  type
  type , 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.

11.11 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. Every symbol, including variables, functions, and types, are associated with some module.

Module definitions are described in §12.1. A program consists of one or more modules. The execution of a program and command-line arguments are described in §12.2. Module uses and explicit naming of symbols is described in §12.3. Nested modules are described in §12.4. The relation between files and modules is described in §12.5.

12.1 Module Definitions

A module is declared with the following syntax:

module-declaration-statement:
  module *identifier block-statement*

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.

Module declaration statements may only be top-level statements in files or top-level statements in other modules. A module that is declared in another module is called a nested module (§12.4).

12.2 Program Execution

Chapel programs start by executing the main function (§12.2.1). The main function takes no arguments but command-line arguments can be passed to a program via a global array of strings called `argv` (§12.2.2). Command-line flags can be passed to a program via configuration variables, as discussed in §8.5.

12.2.1 The *main* Function

The main function must be called `main` and must have zero arguments. It can be specified with or without parentheses. There can be only one main function in all of the modules that make up a program. Every main function starts by using the module that it is defined in, and thus executing the top-level code in that module (§12.2.3).

The main function can be omitted if there is only a single module in the program other than the standard modules, as discussed in §12.2.4.

12.2.2 Command-Line Arguments

A predefined array of strings called `argv` is used to capture arguments to the execution of a program. The number of arguments passed to the program execution can be queried with the array `numElements` function as in

```
argv.numElements
```

> *Status note*. There is no support for the variable `argv`. Only configuration variables allow arguments to be passed to the execution of a program.

12.2.3 Module Execution

Top-level code in a module is executed the first time that module is used via a *use-statement*.

12.2.4 Programs with a Single Module

To aid in exploratory programming, if a program is defined in a single module that uses only standard modules, the module need not define a main function. In this case, a default main function is created to execute the module code.

> *Example*. The code
>
> ```
> writeln("Hello World!");
> ```
>
> is a legal and complete Chapel program. The module declaration is taken to be the file as described in §12.5.

12.3 Using Modules

Modules can be used by code outside of that module. This allows access to the symbols in the modules without the need for explicit naming (§12.3.1). Using modules is accomplished via the use statement as defined in §11.9.

12.3.1 Explicit Naming

All symbols can be named explicitly with the following syntax:

module-named-identifier:
  module-identifier . *identifier*

module-identifier:
  identifier

This allows two variables that have the same name to be distinguished based on the name of their module. If code requires using symbols from two different modules that have the same name, explicit naming is needed to disambiguate between the two symbols. Explicit naming can also be used instead of using a module.

12.4 Nested Modules

A nested module is a module that is defined inside 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.

Example. A nested module can be used without using the outer module by explictly naming the module in the use statement. The code

```
use libmsl.blas;
```

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

12.5 Implicit Modules

Multiple modules can be defined in the same file and do not need to bear any relation to the file in terms of their names. As a convenience, a module declaration statement can be omitted if it is the sole module defined in a file. In this case, the module takes its name from the file.

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 §20 and §14.4.

13.1 Function Definitions

Functions are declared with the following syntax:

function-declaration-statement:
  def *function-name argument-list*$_{opt}$ *var-param-clause*$_{opt}$
    return-type$_{opt}$ *where-clause*$_{opt}$ *block-level-statement*

function-name:
  identifier
  operator-name

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

argument-list:
  (*formals*$_{opt}$)

formals:
  formal
  formal , *formals*

formal:
  formal-tag identifier formal-type$_{opt}$ *default-expression*$_{opt}$
  formal-tag identifier formal-type$_{opt}$ *variable-argument-expression*

formal-type:
  : *type*
  : *TQUESTION identifier*

default-expression:
  = *expression*

variable-argument-expression:
  ... *expression*
  ... *TQUESTION identifier*

formal-tag: *one of*
  in out inout param type

var-param-clause:
  var
  param

where-clause:
  where *expression*

Operator overloading is supported in Chapel on the operators listed above under operator name. Operator and function overloading is discussed in §13.8.

The intents `in`, `out`, and `inout` are discussed in §13.5. The formal tags `param` and `type` make a function generic and are discussed in §21. If the formal argument's type is elided, generic, or prefixed with a question mark, the function is also generic and is discussed in §21.

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.

Functions do not require parentheses if they have no arguments. Such functions are described in §13.10.

Functions can take a variable number of arguments. Such functions are discussed in §13.12.

The optional *var-param-clause* defines a variable function, discussed in §13.6, or a parameter function, discussed in §13.7.

The optional where clause is only applicable if the function is generic. It is discussed in §21.4.

13.2 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;
```

13.3 Function Calls

Functions are called in call expressions described in §10.3. The function that is called is resolved according to the algorithm described in §13.9.

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) result in potential assignments to temporary variables during a function call. For example, passing an array by intent `in`, a temporary array will be created.

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) { ... }
>
> 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 callsite 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 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) { ... }
>
> 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)`. Note that named arguments are necessary to pass actual arguments to formal arguments but use default values for arguments that are specified earlier 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. Domains cannot be passed to a function via the `in` intent.

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. Domains cannot be passed to a function via the `out` intent.

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. Domains cannot be passed to a function via the `inout` intent. The formal argument can be assigned to and read from within the function.

13.6 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. 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 behaviour 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");
  return if i == 0 then x else 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, the line

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

13.7 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 and so is required to be a parameter.

.

13.8 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 §13.9.

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	`+ - * / % ** ! == <= >= < > << >> & | ^`

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

13.9 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. A visible function is any function with the same name that satisfies the criteria in §13.9.1.
- From the set of visible functions, determine the set of candidate functions. A function is a candidate if the function is a valid application of the function call's actual arguments as determined in §13.9.2. A compiler error occurs if there are no candidate functions.
- 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. If there is no function that is more specific than every other candidate function, the function call is ambiguous and a compiler error occurs. The term *more specific function* is defined in §13.9.3.

.

13.9.1 Identifying Visible Functions

A function is a visible function to a function call if the name of the function is the same as the name of the function call and the function is defined or used in a lexical outer scope.

Additionally, functions that have arguments of class type are considered globally visible and so are always visible regardless of the location of their definition.

13.9.2 Determining Candidate Functions

A function is 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 A function call is mapped to a function according to the following steps:

- 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.

13.9.3 Determining More Specific Functions

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

- 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 F_1 shadows F_2, then F_1 is more specific.
- Otherwise, F_1 is not more specific than F_2.

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 and $F1$ is an instantiated parameter, M_1 is more specific.
- If T_{F1} and T_{F2} are the same type and $F2$ is an instantiated parameter, M_2 is more specific.
- If M_1 requires scalar promotion and M_2 does not require scalar promotion, M_2 is more specific.
- If M_2 requires scalar promotion and M_1 does not require scalar promotion, 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 $F2$ is generic over all types and $F1$ is not generic over all types, M_1 is more specific.
- If T_{F1} and T_{F2} are the same type, neither mapping is more specific.
- If T_A and T_{F1} are the same type, M_1 is more specific.
- If 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.

13.10 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 { }
def 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`.

13.11 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 scope in which they are defined. An exception is to a function that has an argument that is a class type. Such functions are globally visible.

13.11.1 Accessing Outer Variables

Nested functions may refer to variables defined in the function in which they are nested. If the function has class arguments, and is thus globally visible, it is a compiler error to refer to a variable in the outer function.

Rationale. It may be too strict to make this a compiler error. Are there advantages to making this a runtime error?

13.12 Variable Length Argument Lists

Functions can be defined to take a variable number of arguments. This allows the call site to pass a different number of actual arguments.

If the variable argument expression is an identifier prepended by a question mark, the number of arguments is variable. Alternatively, the variable expression can evaluate to an integer parameter value requiring the call site to pass that number of arguments to the function.

In the function, the formal argument is a tuple of the actual arguments.

Example. The code

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

defines a function called `mywriteln` that takes a variable number of arguments 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 function can be made generic (§21) to take arguments of different types by eliding the type.

A tuple of variables arguments can be passed to a function that takes variable arguments by destructuring the tuple in a tuple destructuring expression. The syntax of this expression is given by

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

In this expression, the tuple defined by *expression* is expanded in place to represent its components. This allows for the forwarding of variable arguments as variable arguments.

14 Classes

Classes are an abstraction of a data structure where the storage location is allocated independent of the scope of the variable of class type. Each call to the constructor creates a new data object and returns a reference to the object. Storage is reclaimed automatically as described in §14.9.

14.1 Class Declarations

A class is defined with the following syntax:

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

class-inherit-expression-list:
  class-type
  class-type , inherit-expression-list

class-statement-list:
  class-statement
  class-statement class-statement-list

class-statement:
  type-declaration-statement
  function-declaration-statement
  variable-declaration-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 §14.7.

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.

If a class contains a type alias or a parameter, the class is generic. Generic classes are described in §21.

14.2 Class Assignment

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

14.3 Class Fields

Variables and constants declared within class declarations define fields within that class. (Parameters make a class generic.) 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`.

14.3.1 Class Field Accesses

The field in a class is accessed via a member access expression as described in §10.3.2. Fields in a class can be modified via an assignment statement where the left-hand side of the assignment is a member access.

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`.

14.4 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.

14.4.1 Class Method Declarations

Methods are declared with the following syntax:

method-declaration-statement:
  def *type-binding function-name argument-list*$_{opt}$ *var-clause*$_{opt}$
    return-type$_{opt}$ *where-clause*$_{opt}$ *block-level-statement*
  iterator *type-binding function-name argument-list*$_{opt}$ *var-clause*$_{opt}$
    return-type$_{opt}$ *where-clause*$_{opt}$ *block-level-statement*

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.

14.4.2 Class Method Calls

A method is called by using the member access syntax as described in §10.3.2 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()`.

14.4.3 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) { }
```

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

14.4.4 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);
  }
}
```

14.5 Class Constructors

A class constructor is defined by declaring a method with the same name as the class. The constructor is used to create instances of the class. When the constructor is called, memory is allocated to store a class instance.

14.5.1 The Default Constructor

A default constructor is automatically created for every class in the Chapel program. This constructor is defined such that it has one argument for every field in the class. Each of the arguments has a default value.

The default constructor is very useful but its generality in terms of having one argument for each field all of which have default values makes it slightly difficult for the user to create their own constructor. It is expected that in many simple cases, the default constructor will be all that is necessary.

Example. Given the class

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

then instances of the class can be created by calling the default constructor as follows:

- The call `C()` is equivalent to `C(0,3.14,"Hello, World`”)!.
- The call `C(2)` is equivalent to `C(2,3.14,"Hello, World`”)!.
- The call `C(z="")` is equivalent to `C(0,3.14,"")`.
- The call `C(0,0.0,"")` is equivalent to `C(0,0.0,"")`.

14.6 Variable Getter Methods

All field accesses are resolved via getters that are variable methods (§13.6) 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.

14.7 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.

14.7.1 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 §14.7.4.

14.7.2 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.

14.7.3 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.

14.7.4 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.

14.7.5 Inheriting from Multiple Classes

Status note. Multiple inheritance is not yet supported.

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.

14.8 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.

14.9 Automatic Memory Management

> *Status note*. Memory allocated to store class objects is not yet reclaimed.

Memory associated with class instances is reclaimed automatically when there is no way for the current program to reference this memory. The programmer does not need to free the memory associated with class instances.

15 Records

A record is a data structure that is like a class but has value semantics. The key differences between records and classes are described in this section.

15.1 Record Declarations

A record is defined with the following syntax:

> *record-declaration-statement*:
>   **record** *identifier inherit-type-list*$_{opt}$ {
>   *record-statement-list* }
>
> *record-inherit-expression-list*:
>   *record-type*
>   *record-type* , *inherit-expression-list*
>
> *record-statement-list*:
>   *record-statement*
>   *record-statement record-statement-list*
>
> *record-statement*:
>   *type-declaration-statement*
>   *function-declaration-statement*
>   *variable-declaration-statement*

The only difference between record and class declarations is that the `record` keyword replaces the `class` keyword.

15.2 Class and Record Differences

The main differences between records and classes are that records are value classes. They do not need to be reclaimed since the data is recovered when the variable goes out of scope. Records do not support dynamic dispatch and are assigned by value.

15.2.1 Records as Value Classes

A record is not a reference to a storage location that contains the data in the record but is more like a variable of a primitive type. A record directly contains the data associated with the fields in the record.

15.2.2 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.

15.2.3 Record Assignment

In record assignment, the fields of the record on the left-hand side of the assignment are assigned the values in the fields of the record on the right-hand side of the assignment. When a base record is assigned a derived record, just the fields that exist in the base record are assigned. Record assignment is generic in that the right-hand side expression can be of any type as long as the type contains the same fields (by name) as the record on the left-hand side.

15.3 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.

16 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.

16.1 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*

16.1.1 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.

16.2 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.

16.3 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.

> *Status note*. The type-select statement is not yet implemented on unions.

17 Tuples

A tuple is an ordered set of components that allows for the specification of a light-weight record with anonymous fields.

17.1 Tuple Expressions

A tuple expression is a comma-separated list of expressions that is enclosed in parentheses. The number of expressions is the size of the tuple and the types of the expressions determine the component types of the tuple.

The syntax of a tuple expression is given by:

tuple-expression:
(*expression-list*)

expression-list:
expression
expression , *expression-list*

Example. The statement

```
var x = (1, 2);
```

defines a variable `x` that is a 2-tuple containing the values `1` and `2`.

17.2 Tuple Type Definitions

A tuple type is a comma-separated list of types. The number of types in the list defines the size of the tuple, which is part of the tuple's type. The syntax of a tuple type is given by:

tuple-type:
(*type-list*)

type-list:
type
type , *type-list*

Example. Given a tuple expression `(1, 2)`, the type of the tuple value is `(int, int)`, referred to as a 2-tuple of integers.

17.3 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. The assignments are simultaneous so that each component expression on the right-hand side is fully evaluated before being assigned to the left-hand side.

17.4 Tuple Destructuring

When a tuple expression appears on the left-hand side of an assignment statement, the expression on the right-hand side is said to be *destructured*. The components of the tuple on the right-hand side are assigned to each of the component expressions on the left-hand side. This assignment is simultaneous in that the right-hand side is evaluated before the assignments are made.

Example. Given two variables of the same type, x and y, they can be swapped by the following single assignment statement:

```
(x, y) = (y, x);
```

17.4.1 Variable Declarations in a Tuple

Variables can be defined in a tuple to facilitate capturing the values from a function that returns a tuple. The extension to the syntax of variable declarations is as follows:

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

tuple-variable-declaration:
  (*identifier-list*) *type-part*$_{opt}$ *initialization-part*
  (*identifier-list*) *type-part*

The identifiers defined within the *identifier-list* are declared to be new variables in the scope of the statement. The *type-part* and/or *initialization-part* defines a tuple that is destructured when assigned to the new variables.

17.4.2 Ignoring Values with Underscore

If an underscore appears as a component in a tuple expression in a destructuring context, the expression on the right-hand side is ignored, though it is still evaluated.

17.5 Homogeneous Tuples

A homogeneous tuple is a special-case of a general tuple where the types of the components are identical. Homogeneous tuples have less restrictions for how they can be indexed (§17.6).

17.5.1 Declaring Homogeneous Tuples

A homogeneous tuple type may be specified with the following syntax if it appears as a top-level type in a variable declaration, formal argument declaration, return type specification, or type alias declaration:

homogeneous-tuple-type:
  integer-parameter-expression $*$ *type*

integer-parameter-expression:
  expression

The homogeneous tuple type specification is syntactic sugar for the type explicitly replicated a number of times equal to the *integer-parameter-expression*.

Example. The following types are equivalent:

```
3*int     (int, int, int)
```

17.6 Tuple Indexing

A tuple may be indexed into by an integer. Indexing a tuple is given by the following syntax:

tuple-indexing-expression:
  expression (*integer-expression*)

The result of indexing a tuple by integer k is the value of the kth component. If the tuple is not homogeneous, the tuple can only be indexed by an integer parameter. This ensures that the type of the indexing expression is known at compile-time.

17.7 Formal Arguments of Tuple Type

Status note. Formal arguments of tuple type are treated as if they were records. Conversions are not applied to the components.

17.7.1 Formal Argument Declarations in a Tuple

Status note. Formal arguments cannot be grouped together in a tuple.

18 Ranges

Ranges contain a sequence of values of integral type that can be specified with a low bound l, a high bound h, and a stride s. If the stride is negative, the values contained by the range are $h, h-s, h-2s, h-3s, ...$ such that all of the values in the sequence are greater than l. If the stride is positive, the values contained by the range are $l, l+s, l+2s, l+3s, ...$ such that all of the values in the sequence are less than h.

A range is specified by the syntax

> *range-literal*:
> *expression* .. *expression*

The first expression is taken to be the lower bound, the second expression is taken to be the upper bound. The stride of the range is 1 and can be modified with the `by` operator.

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 `uint64`. The type is determined by conceptually adding the low and high bounds together.

18.1 Strided Ranges

The `by` operator can be applied to any range to create a strided range. It is predefined over a range and an integer to yield a new range that is strided by the integer. Striding of strided ranges is accomplished by multiplying the strides.

18.2 Range Assignment

18.3 Querying the Bounds and Stride of a Range

```
def range.low: eltType
def range.high: eltType
def range.stride: int
```

These routines respectively return the low bound, the high bound, and the stride of the range. The type of the returned low and high bound is the element type of the range.

18.4 Unbounded Ranges

An unbounded range is specified by the syntax

> *unbounded-range-literal*:
> *expression* ..
> .. *expression*

Unbounded ranges can be iterated over with zipper iteration 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 zip an unbounded range with a range that does not have the same sign stride.

Unbounded ranges can be used to index into ranges, domains, arrays, and strings. In thesse cases, the elided bound conforms to the expression being indexed.

Status note. Indexing by unbounded ranges is not yet supported.

19 Domains and Arrays

A *domain* is a description of a collection of names for data. These names are referred to as the *indices* of the domain. All indices for a particular domain are values with some common type. Valid types for indices are primitive types and class references or unions, tuples or records whose fields are valid types for indices. This excludes ranges, domains, and arrays. Domains have a rank and a total order on their elements. An *array* is generically a function that maps from a *domain* to a collection of variables. Chapel supports a variety of kinds of domains and arrays defined over those domains as well as a mechanism to allow application specific implementations of arrays.

Arrays abstract mappings from sets of values to variables. This key use of data structures coupled with the generic syntactic support for array usage increases software reusability. By separating the sets of values into their own abstraction, domains, distributions can be associated with sets rather than variables. This enables the orthogonality of data distributions. Distributions are discussed in §23.

19.1 Domains

Domains are first-class ordered sets of indices. There are five kinds of domains:

- Arithmetic domains are rectilinear sets of Cartesian indices that can have an arbitrary rank.
- Sparse domains are subsets of indices in arithmetic domains.
- Indefinite domains are sets of indices where the type of the index is some type that is not an array, domain, or range. Indefinite domains define dictionaries or associative arrays implemented via hash tables.
- Opaque domains are sets of anonymous indices. Opaque domains define graphs and unspecified sets.
- Enumerated domains are sets of constants defined by some enumerated type.

19.1.1 Domain Types

Domain types vary based on the kind of the domain. The type of an arithmetic domain is parameterized by the rank of the domain and the integral type of the indices. The type of a sparse domain is parameterized by the type of the arithmetic domain that defines the superset of its indices. The type of an indefinite domain is parameterized by the type of the index. The type of an opaque domain is unique. The type of an enumerated domain is parameterized by the enumerated type.

Example. In the code

```
var D: domain(2) = [1..n, 1..n];
```

`D` is defined as a two-dimensional arithmetic domain 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 $j \in 1, 2, \ldots, n$.

19.1.2 Index Types

Each domain has a corresponding *index* type which is the type of the domain's indices qualified by its status as an index. Variables of this type can be declared using the following syntax:

index-type:
  index (*domain-expression*)

If the type of the indices of the domain is `int`, then the index type can be converted into this type.

A value with a type that is the same as the type of the indices in a domain but is not the index type can be converted into the index type using a special "method" called `index`.

Example. In the code

```
var j = D.index(i);
```

the type of the variable `j` is the index type of domain `D`. The variable `i`, which must have the same type as the underlying type of the indices of `D`, is verified to be in domain `D` before it is assigned to `j`.

Values of index type are known to be valid and may have specialized representations to facilitate accessing arrays defined for that domain. It may therefore be less expensive to access arrays using values of appropriate index type rather than values of the more general type the domain is defined over.

Status note. In the current implementation, the index type is not distinguished from the underlying type of the indices. The index method is not yet implemented.

19.1.3 Domain Assignment

Domain assignment is by value. If arrays are declared over a domain, domain assignment impacts these arrays as discussed in §19.8, but the arrays remain associated with the same domain regardless of the assignment.

19.1.4 Formal Arguments of Domain Type

Domains are passed to functions by reference. Formal arguments that receive domains are aliases of the actual arguments. It is a compile-time error to pass a domain to a formal argument that has a non-blank intent.

19.1.5 Iteration over Domains

All domains support iteration via forall and for loops over the indices in the set that the domain defines. The type of the indices returned by iterating over a domain is the index type of the domain.

19.1.6 Domain Promotion of Scalar Functions

Domain promotion of a scalar function is defined over the domain type and the type of the indices of the domain (not the index type).

> *Example.* Given an array `A` with element type `int` declared over a one-dimensional domain `D` with integral type `int`, then the array can be assigned the values given by the indices in the domain by writing
>
> ```
> A = D;
> ```

19.2 Arrays

Arrays associate variables or elements with the sets of indices in a domain. Arrays must be declared over domains and have a specified element type.

19.2.1 Array Types

The type of an array is parameterized by the type 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*

domain-expression:
  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 square brackets around the domain literal can be omitted.

> *Example.* In the code
>
> ```
> var A: [D] real;
> ```
>
> `A` is declared to be an array over domain `D` with elements of type `real`.

> *Status note.* Arrays of arrays are not currently supported.

19.2.2 Array Indexing

Arrays can be indexed by indices in the domain they are declared over. The indexing results in an access of the element that is mapped by this index.

Example. If `A` is an array with element type `real` declared over a one-dimensional arithmetic domain `[1..n]`, then the first element in `A` can be accessed via the expression `A(1)` and set to zero via the assignment `A(1) = 0.0`.

Indexing into an array with a domain is call array slicing and is discussed in the next section.

Arithmetic arrays also support indexing over the components of their indices for multidimensional arithmetic domains (where the indices are tuples), as described in §19.3.5.

19.2.3 Array Slicing

An array can be indexed by a domain that has the same type as the domain which the array was declared over. Indexing in this manner has the effect of array slicing. The result is a new array declared over the indexing domain where the elements in the array alias the elements in the array being indexed.

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`.

Arithmetic arrays also support slicing by indexing into them with ranges or tuples of ranges as described in §19.3.6.

19.2.4 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 (i,e) in (A.domain,B) do
  A(i) = e;
```

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 elementwise.

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 i in A.domain do
  A(i) = e;
```

19.2.5 Formal Arguments of Array Type

Arrays are passed to functions by reference. Formal arguments that receive arrays are aliases of the actual arguments. The ordinary rule that disallows assignment to formal arguments of blank intent does not apply to arrays.

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 §21.1.6.

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; it is described in §19.3.7.

19.2.6 Iteration over Arrays

All arrays support iteration via forall and for loops over the elements mapped to by the indices in the array's domain.

19.2.7 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.

Status note. In the current implementation, promotion always returns one-dimensional arrays.

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`.

19.2.8 Array Initialization

By default, the elements in an array are initialized to the default values associated with the element type of the array. There is an expectation that this default initialization can be overridden for performance reasons by explicitly marking the array type or variable.

The initialization expression in the declaration of an array can be based on the indices in the domain using special array declaration syntax that replaces both the type and initialization specifications of the declaration:

special-array-declaration:
  identifier-list indexed-array-type-part initialization-part

indexed-array-type-part:
  : *array-type-forall-expression type*

array-type-forall-expression:
  [*identifier* **in** *domain-expression*]

initialization-part:
  = *expression*

In this code, the *array-type-forall-expression* is syntactic sugar for surrounding the *initialization-part* with this basic forall-expression.

Given a domain expression `D`, an element type `t`, an expression `e` that is of type `t` or that can be implicitly converted to type `t`, then the declaration of array `A` given by

```
var A: [i in D] t = e;
```

is equivalent to

```
var A: [D] t = [i in D] e;
```

The scope of the forall expression is as in the rewritten part so the expression `e` can include references to index `i`.

19.2.9 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:
  var *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)`.

19.3 Arithmetic Domains and Arrays

An arithmetic domain is a rectilinear set of Cartesian indices. Arithmetic domains are specified as a tuple of ranges enclosed in square brackets.

19.3.1 Arithmetic Domain Literals

An arithmetic domain literal is specified by the following syntax:

arithmetic-domain-literal:
  [*range-expression-list*]

range-expression-list:
  range-expression
  range-expression , *range-expression-list*

range-expression:
  expression

Example. The expression `[1..5, 1..5]` defines a 5×5 arithmetic domain with the indices $(1,1),(1,2),\ldots,(5,5)$.

19.3.2 Arithmetic Domain Types

The type of an arithmetic domain is determined from the rank of the arithmetic domain (the number of ranges that define it) and by an underlying integeral type called the *dimensional index type* which must be identical to each of the integral element types of the ranges that define the arithmetic domain. By default, the dimensional index type of an arithmetic domain is `int`.

The arithmetic domain type is specified by the syntax of a function call to the keyword `domain` that takes at least an argument called `rank` that is a parameter of type `int` and optionally an integral type named `dim_type`.

> *Example*. The expression `[1..5, 1..5]` defines an arithmetic domain with type `domain(2,int)`.

19.3.3 Strided Arithmetic Domains

If the ranges that define an arithmetic domain are strided, then the arithmetic domain is said to be strided.

The `by` operator can be applied to any arithmetic domain to create a strided arithmetic domain. It is predefined over an arithmetic domain and an integer or a tuple of integers. In the integer case, the ranges in each dimension are strided by the integer. In the tuple of integers case, the size of the tuple must match the rank of the domain; the integers stride each dimension of the domain. If the ranges are already strided, the strides applied by the `by` operator are multiplied to the strides of the ranges.

19.3.4 Arithmetic Domain Indexing

Arithmetic domains support indexing by a value of type `int` that is at least one and no more than the rank of the array. Indexing into an arithmetic domain returns the range associated with that dimension.

Example. In the code

```
for i in D(1) do
  for j in D(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.

19.3.5 Arithmetic Array Indexing

In addition to being indexed by indices defined by their arithmetic domains, arithmetic arrays can be indexed directly by values of the dimensional index type where the number of values is equal to the rank of the array. This has the semantics of first moving the values into a tuple and then indexing into the array.

Example. Given the definition

```
var ij = (i,j);
```

the indexing expressions `A(ij)` and `A(i,j)` are equivalent.

19.3.6 Arithmetic Array Slicing

In addition to slicing an arithmetic array by an arithmetic domain, arithmetic arrays also support slicing by ranges directly. If each dimension is indexed by an range, this is equivalent to slicing the domain by an arithmetic domain defined by those ranges.

> *Status note*. It is currently required that each dimension be indexed by an range. There is an expectation that indexing some dimensions directly by values of integral type will result in an array slice of a different rank. In the current implementation, doing so would result in a promotion of the array indexing function and would have similar semantics though with more restrictions.

19.3.7 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.

19.4 Sparse Domains and Arrays

> *Status note*. Sparse domains are not yet implemented.

A sparse domain type is given by the syntax

sparse-domain-type:
  sparse domain (*domain-expression*)

domain-expression:
  expression

A sparse domain is a domain that contains a subset of the indices in the domain specified by the *domain-expression*, sometimes called the *base domain*.

Arrays declared over sparse domains can be indexed by all of the indices in the base domain. If the index is not part of the sparse domain, the element returned is called the *unrepresented element*. It is an error to assign a value to the unrepresented element by indexing into the array and assigning it a value. The unrepresented element can be set to any value but by default contains the default value associated with the element type of the array.

19.4.1 Changing the Indices in Sparse Domains

Indices can be added to or removed from sparse domains. Sparse domains support a method `add` that takes an index and adds this index to the sparse domain. All arrays declared over this sparse domain can now be assigned values corresponding to this index.

Sparse domains support a method `remove` that takes an index and removes this index from the sparse domain. The values in the arrays indexed by the removed index are lost.

The operators `+=` and `-=` have special semantics for sparse domains; they are interpreted as calls to the `add` and `remove` methods respectively. The statement

```
D += i;
```

is equivalent to

```
D.add(i);
```

Similarly, the statement

```
D -= i;
```

is equivalent to

```
D.remove(i);
```

19.5 Indefinite Domains and Arrays

An indefinite domain type can be defined over any scalar type and is given by the following syntax:

sparse-domain-type:
  domain (*scalar-type*)

scalar-type:
  type

A scalar type is any primitive type, tuple of scalar types, or class, record, or union where all of the fields have scalar types. Enumerated types are scalar types but domains declared over enumerated types are described in §19.7. Arrays declared over indefinite domains are dictionaries mapping from values to variables.

19.5.1 Changing the Indices in Indefinite Domains

Like with sparse domains, indices can be added or removed to indefinite domains. Indefinite domains support a method `add` that takes an index and adds this index to the indefinite domain. All arrays declared over this indefinite domain can now access elements corresponding to this index.

Indefinite domains support a method `remove` that takes an index and removes this index from the indefinite domain. The values in the arrays indexed by the removed index are lost.

The operators `+=` and `-=` have special semantics for indefinite domains; they are interpreted as calls to the `add` and `remove` methods respectively. The statement

```
D += i;
```

is equivalent to

```
D.add(i);
```

Similarly, the statement

```
D -= i;
```

is equivalent to

```
D.remove(i);
```

19.5.2 Testing Membership in Indefinite Domains

An indefinite domain supports a `member` method that can test whether a particular value is part of the index set. It returns `true` if the index is in the indefinite domain and otherwise returns `false`.

19.6 Opaque Domains and Arrays

Status note. Opaque domains are not yet implemented.

An opaque domain is a form of indefinite domain where there is no algebra on the types of the indices. The indices are, in essence, opaque. The opaque domain type is given by the following syntax:

opaque-domain:
  opaque domain

New index values for opaque domains are explicitly requested via a method called `new`. Indices can be removed by a method called `remove`.

Opaque domains permit more efficient implementations than indefinite domains under the assumption that creation of new domain index values is rare.

19.7 Enumerated Domains and Arrays

Status note. Enumerated domains are not yet implemented.

Enumerated domains are a special case of indefinite domains where the indices are the constants defined by an enumerated type. Enumerated domains do not support the `add` or `remove` methods. All of the constants defined by the enumerated type are indices into the enumerated domain.

An enumerated domain is specified as an indefinite domain would be except the type is an enumerated type rather than some other scalar type.

19.8 Association of Arrays to Domains

When an array is declared, it is linked during execution to the domain over which it was declared. This linkage is constant 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 default value associated with the element type. In the case of removing an index, the element in the array is removed.

When a domain is reassigned a new value, the array is also impacted. Values that could be indexed by both the old domain and the new domain are preserved in the array. 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 and initialized to the default value associated with their type.

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.

19.9 Subdomains

Status note. Subdomains are not yet implemented.

A subdomain is a domain whose indices are indices of a *base domain*. A subdomain is specified by the following syntax:

subdomain-type:
  domain (*domain-expression*)

The ordering of the indices in the subdomain is consistent with the ordering of the indices in the base domain.

Subdomains are verified during execution even as domains are reassigned. The indices in a subdomain are known to be indices in a domain, allowing for fast bounds-checking.

In the case of arithmetic domains, the subdomain literal may be composed of indefinite ranges. In such cases, the omitted bounds of the indefinite ranges are taken from the bounds of the base domain.

19.10 Predefined Functions and Methods on Domains

There is an expectation that this list of predefined functions and methods will grow.

```
def Domain.numIndices: dim_type
```

Returns the number of indices in the domain.

19.11 Predefined Functions and Methods on Arrays

There is an expectation that this list of predefined functions and methods will grow.

```
def Array.numElements: this.domain.dim_type
```

Returns the number of elements in the array.

20 Iterators

An iterator is a function that conceptually returns multiple values rather than simply a single value.

20.1 Iterator Functions

The syntax of an iterator declaration is identical to that of a function declaration except that the keyword `def` is replaced with the keyword `iterator`. The body of the iterator may include yield statements alongside return 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.

20.2 The Yield Statement

The yield statements can only appear in iterators. The syntax of the yield statement is given by

yield-statement:
  yield *expression* ;

20.3 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, can be captured in new variable declarations or arrays, in which case they evaluate to an array of values, or can be passed to a generic function argument.

20.3.1 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.

20.3.2 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

```
iterator 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`.

20.3.3 Iterators and Generics

If an iterator call expression is passed to a function argument that is generic, the iterator is passed without being evaluated and is treated as a closure within the generic function.

20.4 Scalar Promotion

A function requires scalar promotion if an iterator (or array, domain, or range) is passed to a formal argument with a type that allows the yielded type of the iterator to dispatch to the formal argument. In the case of arrays, the yielded type is the element type. In the case of domains and ranges, the yielded type is the index type. The rules of when an overloaded function is promoted are discussed in §13.9. 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 an iterator

```
iterator oneToFive() {
  for i in 1..5 do
    yield i;
}
```

and a function

```
def square(x: int) return x**2;
```

then the call `square(oneToFive())` results in the promotion of the `square` function over the values returned by the `oneToFive` iterator. The result is an iterator that returns the values `1`, `4`, `9`, `16`, and `25`. Instead of using the `oneToFive` iterator to promote the `square` function, the range `1..5` could be used directly as in `square(1..5)`. Also note that operator invocations are treated as function calls in terms of promotion so `(1..5)**2` is also equivalent.

20.4.1 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 ”.

20.4.2 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 ”.

21 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.

21.1 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.

21.1.1 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");
```

21.1.2 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`.

21.1.3 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)`.

21.1.4 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;
}
```

21.1.5 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 §21.5) 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`.

The generic types `integral` and `numeric` are generic types that can only be instantiated with, respectively, the signed and unsigned integeral types and all of the numeric types.

21.1.6 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.

21.2 Function Visibility in Generic Functions

Function visibility in generic functions is altered depending on the instantiation. When resolving function calls made within visible functions, the visible functions are taken from any call site at which the function is instantiated for each particular instantiation.

21.3 Generic Types

A class or record is generic if any of the following conditions hold:

- The class contains a specified or unspecified type alias.
- The class contains a field that is a parameter.
- The class contains a field that has no type and no initialization expression.
- The class contains a field where the type of the field is generic.

21.3.1 Type Aliases in Generic Types

Type aliases defined in a class or a record can be unspecified type aliases; type aliases that are not bound to a type. If a class or record contains an unspecified type alias, the aliased type must be specified whenever the type is used.

A type alias defined in a class or record is accessed as if it were a field. Moreover, it becomes an argument with intent `type` to the default constructor for that class or record. This makes the default constructor generic. When the default constructor is instantiated, the type is instantiated where the type bound to the type alias is set to be the type passed to the default constructor.

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 `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.

21.3.2 Parameters in Generic Types

Parameters defined in a class or record do not require an initialization expression. If they do not have an initialization expression, the parameter must be specified whenever the type is used.

A parameter defined in a class or record is accessed as if it were a field. This access returns a parameter. Parameters defined in classes or records become arguments with intent `param` to the default constructor for that class or record. This makes the default constructor generic. When the default constructor is instantiated, the type is instantiated where the parameter is bound to the parameter passed to the default constructor.

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 `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.

21.3.3 Fields without Types

If a 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 must be specified when the class or record is constructed or specified. The field becomes an argument to the default constructor that has no specified type and no default value. This makes the default constructor generic. When the default constructor is instantiated, the type is instantiated where the type of the field becomes the type of the actual argument passed to the default constructor.

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) = nil;
}
```

A node with integer element type can be defined in the call to the constructor. The call `Node(1)` defines a node with the value `1`. The code

```
var list = Node(1);
list.next = Node(2);
```

defines a two-element list with nodes containing the values `1` and `2`.

21.3.4 Fields of Generic Types

If a field in a class or record is specified to have a generic type, then the class or record is generic over the type of this field and the type of the field is constrained to be an instantiation of the field's specified generic type.

21.3.5 Generic Methods

All methods bound to generic classes or records are generic over the implicit `this` argument and any other argument that is generic.

21.3.6 The *eltType* Type

The common idiom of parameterizing a collection-oriented data type by a single element type has special syntactic support given by

> *of-type*:
>   *type* **of** *type*

This syntax is a short-hand for passing the second type by name `eltType` as the only argument to the first type. Given the definition of `Node` in the example in §21.3.1, one can specify the type `Node(real)` or `Node(eltType=real)` by writing `Node of real`.

21.4 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 (§13.1). 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 { ... }
def foo(x) where x.type == real { ... }
```

the call foo(3) resolves to the first definition because when the second function is instantiated the where clause evaluates to false.

21.5 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 = 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;
}
```

22 Parallelism and Synchronization

Chapel is an explicitly parallel programming language. Parallelism is introduced into a program by the user with the following three constructs: `forall`, `cobegin`, and `begin`. In addition, some operations on arrays and domains, as well as invocations of promotion, are executed in parallel. Synchronization is provided with *synchronization variables* and *atomic* statements. To avoid any unintended implications, the terms *computation* and *sub-computation* will be used to refer to distinct, concurrently executing portions of the program.

22.1 The Forall Loop

The forall loop is a variant of the for loop that allows for the concurrent execution of the loop body. The for loop is described in §11.8. The syntax for the forall loop is given by

forall-statement:
  forall *index-expression* **in** *iterator-expression* **do** *statement*
  forall *index-expression* **in** *iterator-expression block-level-statement*

The forall loop 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 compiler and runtime determine the actual concurrency based on the specification of the iterator of the loop. The keyword `ordered`, described in §22.1.2, can be used to constrain the parallelism to give a partial order on the iterator.

Control continues with the statement following the forall loop only after each iteration has been completely evaluated. Control transfers out of a loop body via `break`, `continue`, and `return` are not permitted. Control can be transferred out of the loop via a `yield` statement.

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 performed serially, with maximum concurrency where each loop body iteration instance is executed in a separate computation, or somewhere in between.

Status note. The forall loop is currently executed serially.

22.1.1 Alternative Forall Loop Syntax

The forall loop may be alternatively specified with a more concise syntax given by:

alternative-forall-statement:
  [*index-expression* **in** *iterator-expression*] *statement*

The semantics are unchanged.

Example. The previous `forall` example can be alternatively written as:

```
[i in 1..N] a(i) = b(i);
```

22.1.2 The Ordered Forall Loop

By default a forall loop allows complete concurrent evaluation of the iterator expression and among the loop instances. The keyword `ordered` can be used to constrain the general parallelism among instances of the loop to that expressed by an iterator. This allows an iterator to both define an array of values and to impose a partial order on that iterator. This has the same semantics as with the ordered expression which is explained in §22.5. The syntax is:

ordered-forall-statement:
  ordered forall *index-expression* **in** *iterator-expression* **do** *statement*
  ordered forall *index-expression* **in** *iterator-expression block-level-statement*

Example. In the code

```
ordered forall i in walk(root) do
  work(i);

iterator walk(n: node) {
  yield n;
  forall c in 0..n.numOfChildren {
      yield n.child[c];
  }
}
```

there is a contraint on the parallel execution such that the function `work` is evaluated on a node before any of its immediate children nodes. The work on sibling nodes can be executed concurrently.

Status note. The ordered forall loop is currently executed serially.

22.2 The Forall Expression

With syntax similar to the alternative forall loop statement, a forall expression can be used to enable concurrent evaluation of sub-expressions. The sub-expressions are evaluated once for each element in the iterator expression. The syntax of a forall expression is given by

forall-expression:
  [*index-expression* **in** *iterator-expression*] *expression*

A forall expression is semantically equivalent to an iterator that yields the expressions.

Example. The code

```
[i in S] f(i);
```

is equivalent to

```
iterator ff() {
  for i in S do
    yield f(i);
}
ff();
```

Status note. Forall expressions are evaluated serially.

22.2.1 Filtering Predicates

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.length] if i % 2 == 1 then s(i)
> ```

22.3 The Cobegin Statement

The cobegin statement is used to create parallelism among statements within a block statement. The `cobegin` statement syntax is

> *cobegin-statement*:
>   **cobegin** *block-statement*

Each statement within the block statement is executed concurrently and is considered a separate computation. Control continues after all of the statements within the block statement have been evaluated.

As with the forall loop, control transfers are not permitted either into or out of the cobegin's block statement. Similarly, yield statements are allowed.

Variables declared in the cobegin statement are *single variables*, described in §22.7.1.

22.4 The Begin Statement

The begin statement spawns a computation to execute a statement. Control continues simultaneously with the statement following the begin statement. The begin statement is an unstructured way to create a new computation that is executed only for its side-effects. The syntax for the begin statement is given by

> *begin-statement*:
>   **begin** *statement*

The following statements cannot be contained in begin-statements: break-statements, continue-statements, yield-statements, and return-statements.

22.5 The Ordered Expression

> *Status note.* The ordered expression is not yet implemented.

The `ordered` keyword can be used as an unary operator to suppress parallel execution among instances of an expression that can involve side-effects to memory. The `ordered` keyword does not inhibit parallelism within the sub-expression. The syntax is:

> *ordered-expression*:
>   **ordered** *expression*

Example. In the code

```
ordered [i in S] f(i)
```

f is a function and S is an iterator expression. Each instance of f(i) is executed once for each value in S and in serial order. The ordered constraint does not propagate to inhibit parallelism within f.

22.6 The Serial Statement

The serial statement can be used to dynamically control the degree of parallelism. The syntax is:

serial-statement:
  serial *expression block-level-statement*

where the expression evaluates to a bool type. Independent of that value, the *block-level-statement* is evaluated. If the expression is true, any dynamically encountered forall loop or cobegin statement is executed serially within the current computation. Any dynamically encountered begin-statement is executed serially with the current computation; no new computation is spawned. Control continues to the statement following the begin-statement after the begin-statement finishes.

Example. In the code

```
ordered forall i in walk(root) do
  work(i);

iterator walk(n: node) {
  yield n;
  serial n.depth > 4 forall c in 0..n.numOfChildren {
      yield n.child[c];
  }
}
```

the serial statement inhibits concurrent execution on the tree for nodes that are deeper than four levels in the tree.

There is an expectation that functions that may be executed in a serial context are cloned to avoid the overhead of testing and suppressing parallelism.

22.7 Synchronization Variables

Synchronization variables are used to coordinate computations that share data. The use of and assignment to these variables implicitly controls the execution order of the computation. There are two kinds of synchronization variables, *single* and *sync* variables. A single variable can only be assigned once during its lifetime. A sync variable can be assigned multiple times during its lifetime.

The normal use of and assignment to a synchronization variable is well suited for producer-consumer data sharing. Additional functions on synchronization variable are provided such that other traditional synchronization primitives, such as semaphores and mutexes, can be contructed.

22.7.1 Single Variables

A single (assignment) variable can only be assigned once during its lifetime. A use of a single variable before it is assigned causes the computation's execution to be suspended until the variable is assigned. Otherwise, the use proceeds as with normal variables and the computation continues. After a single assigment variable is assigned, all computations 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*

Example. In the code

```
class Tree {
  var is_leaf : bool;
  var left    : Tree;
  var right   : Tree;
  var value   : int;

  def sum() {
    if (is_leaf) 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 computation created with the begin statement. The computation returning the sum waits on the reading of `x` until it has been assigned.

While a `cobegin` might be a more suitable formulation, this fragment creates an asynchronous computation to compute the sum of the left sub-tree while the main computation continues with the right sub-tree. The final reference to variable x will be delayed until the assignment to x completes and that value will be used as a summand.

When a single variable has an initializer, the evaluation of that initializer is implicitly performed as an asynchronous computation.

Example. The code

```
var x: single int = left.sum;
```

is equivalent to

```
var x: single int;
x = left.sum;
```

Any variable declaration within a cobegin statement is implicitly treated as a single variable for references in other statements of the cobegin statement.

Example. In the code

```
def sum() {
  if (is_leaf) then
    return value;
  var z;
  cobegin {
    var x = left.sum();
    var y = right.sum();
    z = x+y;
  }
  return z;
}
```

the computation with assignment to `z` waits for the other computations to assign to `x` and `y` before it references `x` and `y` in order to assign to `z`. The variables `x` and `y` are impliclty single.

22.7.2 Sync Variables

A sync variable generalizes the single assignment variable to permit multiple assignments to the variable. A sync variable is logically either *full* or *empty*. When it is empty, computations 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 computation waiting on a sync variable, one is non-deterministically selected to use the variable and resume execution. The other computations continue to wait for the next assignment.

If a computation attempts to assign to a sync variable that is full, the computation is suspended and the assignment is delayed. When the sync variable becomes empty, the computation is resumed and the assignment proceeds, transistioning the state back to full. If there are multiple computations 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

22.7.3 Additional Synchronization Variable Functions

Synchronization variables support additional methods that can be used to bypass their semantics to provide new ones. For sync variable `s`, the following functions are defined:

```
writeFE(s, v) // wait for full, assign s=v, and leave empty
writeXF(s, v) // no wait, assign s=v, and leave full
writeXE(s, v) // no wait, assign s=v, and leave empty
readFF(s)     // wait for full, leave full, and return s's value
readXF(s)     // no wait, leave full, and return s's value
readXX(s)     // no wait, leave F/E unchanged, and return s's value
```

For single variables `s` only `readFF` is defined.

22.7.4 Synchronization Variables of Record and Class Types

A variable of record or class type can be a single or sync variable. The semantics of single and sync variables are applied only to the variable and not to accesses of individual fields. A record or class type may have sychronization variable fields to get synchronization semantics on individual field accesses.

22.8 Memory Consistency Model

This section is forthcoming.

22.9 Atomic Statement

Status note. Atomic statements are not yet implemented.

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.

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. If the performance implications of strong atomicity are not tolerable, the semantics of atomic transactions may be revisited, and could become weaker.

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 != null {
    if last.next == obj {
      found = true;
      last.next = object.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 computation 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 computation 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);
```

23 Locality and Distribution

Status note. Programs can currently only run on a single locale. The abstractions described here are not yet implemented.

Chapel provides high-level abstractions that allow programmers to exploit locality by defining the affinity of data and computation. This is accomplished by associating both data objects and computations with abstract *locales*. To provide a higher-level mechanism, Chapel allows a mapping from domains to locales to be specified. This mapping is called a *distribution* and it guides that placement of variables associated with arrays and the placement of subcomputations defined over the domain.

Throughout this section, the term *local* refers to data that is associated with the locale that a computation is running on and *remote* refers to data that is not. We assume that there is some execution overhead associated with accessing data that may be remote compared to data known to be local.

23.1 Locales

A locale abstracts a processor or node in a parallel computer system, or the basic component in the computer system where local memory can be accessed uniformly.

23.1.1 The Locale Type

The identifier `locale` is a primitive type that abstracts a locale as described above. Both data and computations can be associated with a value of locale type. The only operators defined over locales are the equality and inequality comparison operators.

23.1.2 Predefined Locales Array

A predefined configuration variable defines the *execution environment* for a program. This environment is defined by the following definitions:

```
config const numLocales: int;
const Locales: [1..numLocales] locale;
const Global: locale;
```

The environment consists of constants which are fixed when the program begins execution. The variable `Global` holds a special value of `locale` type that can be distinct from the values stored in `Locales`. This value is used to denote an object or computation that has no defined affinity.

When a program starts, a single computation is executing. It is running on the locale given by `Locales(1)`.

23.1.3 Querying the Locale of a Variable

Every variable `v` is associated with some locale which can be queried using the following syntax:

locale-access:
  expression . **locale**

When the *expression* is a class type, the locale is where the object is located rather than where the *expression* may be located.

23.2 Specifying Locales for Computation

When execution is proceeding on some locale, a computation can be associated with a different locale in two ways: via distributions as discussed in §23.3 or with an *on-statement* as discussed below.

23.2.1 On

The on statement controls on which locale a computation or data should be placed. The syntax of the on statement is given by

on-statement:
  on *expression* **do** *statement*
  on *expression* *block-level-statement*

If the *expression* is a value of `locale` type, the *statement* or *block-level-statement* is executed on the locale specified directly by the expression. Otherwise, the expression must be a variable and the locale is taken to be the locale where the variable is located. Execution continues after the `on-statement` after execution of the *statement* or *block-level-statement* completes.

If the locale that the *expression* refers to is equal to `Global`, then the locale is unspecified and is determined by the runtime and/or compiler.

Example. A common idiom is to use `on` in conjunction with `forall` to access an array decomposed over multiple locales. The code

```
forall i in D do on A(i) {
  // some computation
}
```

executes each iteration of the forall loop on the locale where the element of `A(i)` is located.

By default, when new variables and data objects are created, they are created in the locale where the computation is running. This locale can be changed by using the `on` keyword. Variables can be defined within an *on-statement* to define them on a particular locale.

23.2.2 On and Iterators

When a loop iterates over an iterator, on-statements inside the iterator control where the corresponding loop body is executed.

> *Example*. An iterator over a distributed tree might include an iterator over the nodes as defined in the following code:
>
> ```
> class Tree {
>   var left, right: Tree;
>   iterator nodes {
>     on this yield this;
>     if left then
>       forall t in left.nodes do
>         yield t;
>     if right then
>       forall t in right.nodes do
>         yield t;
>   }
> }
> ```
>
> Given this code and a binary tree of type `Tree` stored in variable `tree`, then we can use the nodes iterator to iterate over the tree with the following code:
>
> ```
> forall t in tree.nodes {
>   // body executed on t as specified in nodes
> }
> ```
>
> Here, each instance of the body of the `forall` loop is executed on the locale where the corresponding object `t` is located. This is specified in the `nodes` iterator where the `on` keyword is used. In the case of zipper or tensor product iteration, the location of execution is taken from the first iterator. This can be overridden by explictly using `on` in the body of the loop or by reordering the product of iteration.

23.3 Distributions

A mapping from domain index values to locales is called a *distribution*.

23.3.1 Distributed Domains

A domain for which a distribution is specified is referred to as a *distributed domain*. A domain supports a method, `locale`, that maps index values in the domain to locales that correspond to the domain's distribution.

Iteration over a distributed domain implicitly executes the control computation in the domain of the associated locale. Similarly, when iterating over the elements of an array defined over a distributed domain, the controlled computations are determined by the distribution of the domain. If there are conflicting distributions in product iterations, the locale of the computation is taken to be the first component in the product.

> *Example*. If `D` is a distributed domain, then in the code
>
> ```
> forall d in D {
>   // body
> }
> ```
>
> the body of the loop is executed in the locale where the index `d` maps to by the distribution of `D`.

23.3.2 Distributed Arrays

Arrays defined over a distributed domain will have the element variables stored on the locale determined by the distribution. Thus, if `d` is an index of distributed domain `D` and `A` is an array defined over that domain, then `A(d).locale` is the locale to which `d` maps to according to `D`.

23.3.3 Undistributed Domains and Arrays

If a domain or an array does not have a distributed part, the domain or array is not distributed and exists only on the locale on which it is defined.

23.4 Standard Distributions

Standard distributions include the following:

- The block distribution `Block`
- The cyclic distribution `Cyclic`
- The block-cyclic distribution `BlockCyclic`
- The cut distribution `Cut`

A design goal is that all standard distributions are defined with the same mechanisms that user-defined distributions (§23.5) are defined with.

23.5 User-Defined Distributions

This section is forthcoming.

24 Reductions and Scans

Chapel provides a set of built-in reductions and scans with parallel semantics, a mechanism for defining more reductions and scans with efficient implementations, and syntact support to make reductions and scans easy to use.

24.1 Reduction Expressions

The syntax for a reduction expression is given by:

reduce-expression:
  reduce-operator **reduce** *expression*
  type **reduce** *expression*

reduce-scan-operator: *one of*
  + $*$ && || & | ^ *min max*

The expression on the right-hand side of the reduction can be of any type that can be iterated over.

The built-in reductions are defined in *reduce-scan-operator*. These include, in order, sum, product, logical and, logical or, bitwise and, bitwise or, bitwise exclusive or, minimum, and maximum.

User-defined reductions are specified by preceding the keyword `reduce` by the class type that implements the reduction interface as described in §24.3.

24.2 Scan Expressions

The syntax for a scan expression is given by:

scan-expression:
  reduce-scan-operator **scan** *expression*
  type **scan** *expression*

The expression on the right-hand side of the scan can be of any type that can be iterated over.

The built-in scans are defined in *reduce-scan-operator*. These are identical to the built-in reductions and are described in §24.1.

User-defined scans are specified by preceding the keyword `scan` by the class type that implements the scan interface as described in §24.3.

24.3 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.

25 Input and Output

Chapel provides a built-in `file` type to handle input and output to files using functions and methods called `read`, `write`, and `writeln`.

25.1 The *file* type

The file type 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 `string` 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 strings:

- `"r"` The file can be read.
- `"w"` The file can be written.

> *Status note*. The `style` field is not yet implemented. All input and output is done in text mode. All files must be text files.
>
> There is an expectation that there will be more styles to control the default reading and writing methods.

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`, `write`, and `writeln` for input and output as discussed in §25.5 and §25.6.

25.2 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.

25.3 The *write*, *writeln*, and *read* 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`.

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!");
```

25.4 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.

25.5 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 computations.

25.5.1 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.

25.5.2 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 = 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`.

25.6 The *read* method on files

The `file` type supports a method `read` that takes an arbitrary number of arguments. Each argument is read in turn by calling a method also bound to the `file` type that takes a single argument of that type.

The `file` type also supports an overloaded method `read` that has a single generic type argument. It reads a single value of the specified type from the file and returns it.

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);
```

25.7 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.

26 Standard Modules

This section describes predefined functions that are available to any Chapel program as well as a set of standard modules that, when used, define a set of functions and types available to Chapel programs. The standard modules include the following:

`BitOps`	Bit manipulation routines
`Math`	(*used by default*) Math routines
`Random`	Random number generation routines
`Standard`	(*used by default*) Basic routines
`Time`	Types and routines related to time
`Types`	(*used by default*) Routines related to primitive types

There is an expectation that each of these modules will be extended and that more standard modules will be defined.

26.1 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.

26.2 Math

The module `Math` defines routines for mathematical computations. This module is used by default; there is no need to explictly used 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: complex): real
```

Returns the absolute value of the argument.

```
def acos(x: real): real
```

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
```

Returns the inverse hyperbolic cosine of the argument. It is an error if `x` is less than 1.

```
def asin(x: real): real
```

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
```

Returns the inverse hyperbolic sine of the argument.

```
def atan(x: real): real
```

Returns the arc tangent of the argument.

```
def atan2(y: real, x: real): real
```

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
```

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
```

Returns the cube root of the argument.

```
def ceil(x: real): real
```

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
```

Returns the cosine of the argument.

```
def cosh(x: real): real
```

Returns the hyperbolic cosine of the argument.

```
def erf(x: real): real
```

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
```

Returns the complementary error function of the argument. This is equivalent to `1.0 - erf(x)`.

```
def exp(x: real): real
```

Returns the value of e raised to the power of the argument.

```
def exp2(x: real): real
```

Returns the value of 2 raised to the power of the argument.

```
def expm1(x: real): real
```

Returns one less than the value of e raised to the power of the argument.

```
def floor(x: real): real
```

Returns the value of the argument rounded down to the nearest integer.

```
def lgamma(x: real): real
```

Returns the natural logarithm of the absolute value of the gamma function of the argument.

```
def log(x: real): real
```

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
```

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
```

Returns the natural logarithm of `x+1`.

```
def log2(i: int(?w)): int(w)
def log2(i: uint(?w)): uint(w)
def log2(x: real): real
```

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
```

Returns the rounded integral value of the argument determined by the current rouding direction.

```
def rint(x: real): real
```

Returns the rounded integral value of the argument determined by the current rouding direction.

```
def round(x: real): real
```

Returns the rounded integral value of the argument. Cases halfway between two integral values are rounded towards zero.

```
def sin(x: real): real
```

Returns the sine of the argument.

```
def sinh(x: real): real
```

Returns the hyperbolic sine of the argument.

```
def sqrt(x: real): real
```

Returns the square root of the argument. It is an error if the argument is less than zero.

```
def tan(x: real): real
```

Returns the tangent of the argument.

```
def tanh(x: real): real
```

Returns the hyperbolic tangent of the argument.

```
def tgamma(x: real): real
```

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
```

Returns the nearest integral value to the argument that is not larger than the argument in absolute value.

26.3 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.

class `RandomStream`

Implements a pseudo-random stream of values. Our 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.

const `RandomStream.seed:` **int**`(64)`

The seed value for the random stream. If no seed is specified in the constructor, the millisecond value of the current time is used. The seed value must be an odd integer. If an even integer is supplied, the class constructor will increment it to obtain an odd integer.

def `RandomStream.fillRandom(x:[?D]` **real**`)`

Fill the argument array, `x`, with the next |`D`| values of the pseudo-random stream. Arrays of arbitrary rank can be passed to this routine, causing the 1D stream of values to be mapped to the array elements according to the array's default iteration order. Once our implementation supports distributed arrays, this routine is intended to fill the array's values in parallel.

def `RandomStream.fillRandom(x:[?D]` **complex**`)`

Similar to the previous routine, but for use with arrays of complex values. The elements are filled in the same order as above except that pairs of values from the stream are assigned to each element, the first to the real component, the second to the imaginary. As this module matures, we will support `fillRandom` for arrays of other element types as well.

`SeedGenerator`

A symbol that can be used to generate seed values for the RandomStream class.

`SeedGenerator.clockMS`

Generates a seed value using the milliseconds value from the current time. As this module matures, `SeedGenerator` will support additional mechanisms for generating seed values.

def `fillRandom(x:[], initseed:` **int**`(64))`

A routine provided for convenience to support filling an array `x` with pseudo-random values without explicitly constructing an instance of the `RandomStream` class, 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 and will default to the milliseconds value of the current time if no seed value is provided.

26.4 Standard

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` as well as the rest of the arguments to the call.

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 `length(s:` **string**`):` **int**

Returns the number of characters in the argument `s`.

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.

26.5 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.

`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` `getCurrentTime(unit: 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 = 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()`

Start 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 the computation for `t` seconds.

26.6 Types

`def` `numBits(`**`type`**` t): `**`int`**

Returns the number of bits used to store the values of type `t`. This is implemented for all numeric types and `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.

Index